Automating Thought of Search: A Journey Towards Soundness and Completeness

Daniel Cao
Cornell University
dyc33@cornell.edu

Michael Katz
IBM Research
Michael.Katz1@ibm.com

Harsha Kokel
IBM Research
Harsha.Kokel@ibm.com

Kavitha Srinivas
IBM Research
Kavitha.Srinivas@ibm.com

Shirin Sohrabi
IBM Research
ssohrab@us.ibm.com

Abstract

Language models (LLMs) have recently used for search, primarily as world models that define the space; the use of LLMs for search forgo soundness for the sake of flexibility. A recent work, Thought of Search (ToS), proposed defining the search space with code, having the language models produce that code. ToS requires a *human in the loop*, collaboratively producing a sound successor function and goal test. The result, however, is worth the effort: all the tested datasets were solved with 100% accuracy. In this work, we automate ToS (AutoToS), completely taking the *human out of the loop* of solving planning problems. AutoToS guides the language model step by step towards the generation of sound and complete search components, through feedback from both generic and domain specific unit tests. We achieve 100% accuracy, with minimal feedback iterations, using LLMs of various sizes on all evaluated domains.

1 Introduction

Spurred by the abilities of large language models (LLMs) in natural language tasks, several recent works have studied AI planning with them as a subset of code generation and code refinement. The approaches vary from giving a planning problem to an LLM and asking it to output an entire plan in a single call [40, 23, 33] to asking an LLM to generate a planning model to be given to an automated planner [13, 32, 12]. Between these two extremes, lies a body of work on using language models to plan by performing a combinatorial search [15, 48, 5, 37]. Among these, Thought of Search (ToS) [25] stands out; it uses LLMs to define the search space for the entire domain at once. It is done simply by soliciting 2 crucial search components, a successor function and a goal test. These components are then plugged into a search algorithm, such as Breadth-First Search (BFS) [9].

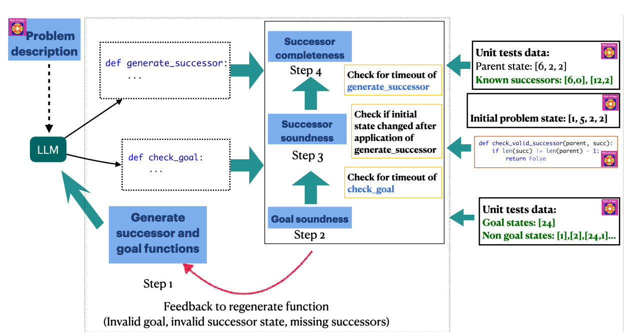

Figure 1: An overview of AutoToS.

ToS has impressive accuracy of 100% on all tested benchmarks and it produces a symbolic model whose soundness and completeness can be verified. However, ToS has a limitation - it requires a

38th Conference on Neural Information Processing Systems (NeurIPS 2024) Workshop on Open-World Agents.

human expert in the loop, providing feedback to the model on the produced code. We automate the iterative feedback and exception handling process through the use of unit tests and printed debugging statements for use with few shot and Chain of Thought (CoT) prompting [6, 45, 27, 44], limiting the human expert involvement with the language model. We test the search components for soundness and completeness and provide feedback to the model when a violation is detected. We use a mixture of domain-independent and domain-specific tests, based on a small number of held out instances.

We exemplify our proposed approach on five representative search problems from the recent literature and test with a variety of LLMs of different sizes. Through automated feedback, we find that the accuracy of the generated code consistently increases to reach 100% across all tested domains. We show that the total number of calls to the LLM is typically small, comparable to the results of ToS with human feedback. In an ablation study, we justify the importance of soundness and completeness feedback for obtaining the highly accurate final code. Finally, we investigate the errors in the code generated by the language models and find that they differ significantly in error distribution.

2 Related Works

Planning with LLMs Recently, several works have leveraged LLMs for plan generation. Valmeekam et al. [43] analyzed LLMs ability to generate plans for classical planning problems described in natural language. Raman et al. [36] generated task plans and used precondition errors as feedback to revise the generated plan. In the same vein, various works have used external verifiers or validators as feedback for LLMs to generate better plans [41, 24]. Pallagani et al. [34] investigate training approaches to improve plan generation abilities. All these approaches use LLMs to solve one problem at a time–essentially treating LLM as a policy. Another line of work has tried to extract policies or generalized plans from LLMs. Silver et al. [39] synthesized generalized plans as Python programs from LLMs for planning domains described in a formal language (PDDL). Further, LLMs have also been used to extract planning problems and models in formal language from their natural language description. Liu et al. [29] used LLMs to translate natural language planning problems to PDDL problems, and Zuo et al. [53] proposed a benchmark for such evaluating this ability while Xie et al. [46] use LLMs to translate natural language goals to PDDL. Recently, Guan et al. [13], Gestrin et al. [12] and Oswald et al. [32] leveraged LLMs to convert natural language domain description to PDDL domains. However, the LLM generated PDDL remains less reliable and difficult to evaluate.

Planning with LLMs using Search A burgeoning research field utilizes LLM's to conduct a search via structured prompting and feedback for planning and reasoning problems. Hao et al. [15] used LLMs in the loop for Monte Carlos Tree search, using LLMs as world models to generate next state and as reasoning agents to pick the next state to expand. Similarly, Tree of Thoughts [49] used LLMs to generate a search tree—to expand each node in the search tree—and also for evaluating the choices and selecting the next best state. Graph of Thoughts [5] modeled LLM generated output as a graph instead of a tree and reduces the number of LLM calls. Similar approaches with integration to search are also proposed for interactive domains [52, 38]. Their significant reliance on LLMs for generating successors makes them not only extremely inefficient but also unsound. Thought of Search (ToS) [25], on the other hand, proposed using LLMs to generate code for the successor and goal functions. Once these functions are available, any offline search algorithm can be used to solve any problem in the domain. This approach is significantly more efficient than approaches which use LLMs in the loop during search. However, it requires human expert for the feedback. Our work focuses on alleviating the requirement of human in the loop feedback.

Code Generation with LLMs LLM's abilities in program synthesis are rapidly advancing. Various benchmarks evaluate correctness of LLM generated code [7, 35, 28]. Subsequent approaches have demonstrated human level performance on coding benchmarks [51, 31]. Execution errors are used as feedback to LLMs to refine the code [8, 50]. External verifies are used to curate feedback for LLMs [30, 19]. Unit test results and error messages were used as feedback [21]. Recently, LLMs code generation has also shown to help in mathematical reasoning problems [51]. Inspired by their success, we automatically feedback ToS with both generic and domain-specific unit tests and validators.

3 Background

We follow the notation of Katz et al. (2018), slightly adapting it for our purposes. A deterministic planning problem over a state space is a tuple $\Pi = \langle S, A, s_0, S_G, f \rangle$, where S is a finite set of *states*, A is a finite set of *action labels*, $s_0 \in S$ is the *initial state*, $S_G \subseteq S$ is the set of *goal states*, and $f : S \times A \to S$ is the *transition function*, such that $f(s, a)$ is the state which applying action a in state s leads to. A triplet $\langle s, a, f(s, a) \rangle$ is called a *transition*. A *solution* to Π is a sequence of states and action labels (trace) $\rho = \langle s_0, a_1, s_1, a_2, \dots a_n, s_n \rangle$, such that $f(s_i, a_{i+1}) = s_{i+1}$ for $0 \leq i < n$ and $s_n \in S_G$. In cases when the action labels are not important, they can be dropped from the definition.

The "black box" approach encodes the state space with a tuple $\Pi_{bb} = \langle s_0, succ, isgoal \rangle$, where s_0 is the initial state, $succ : S \to 2^{A \times S}$ is a successor generator, and $isgoal : S \to \{T, F\}$ is the goal test function. A *solution* to the black-box problem is a sequence of states and action labels (a trace) $\pi = \langle s_0, a_1, s_1, a_2, \dots a_n, s_n \rangle$, such that $\langle a_{i+1}, s_{i+1} \rangle \in succ(s_i)$ for $0 \leq i < n$ and $isgoal(s_n) = T$. Here as well, if action labels are not important, they can be dropped. We now establish the correspondence between the black-box encoding and the planning problem.

Definition 1 (Soundness and completeness) *We say that* $isgoal$ *is* **sound** *if* $isgoal(s) = F$ *for all* $s \notin S_G$ *and* $isgoal$ *is* **complete** *if* $isgoal(s) = T$ *for all* $s \in S_G$*. We say that* $succ$ *is* **sound** *if* $succ(s) \subseteq \{\langle a, s' \rangle \mid f(s, a) = s'\}$ *and* $succ$ *is* **complete** *if* $succ(s) \supseteq \{\langle a, s' \rangle \mid f(s, a) = s'\}$.

Sound and complete successor generator and goal test provide the "black box" description of the state space of the planning problem Π. In such cases, a solution to Π_{bb} is guaranteed to be a solution to Π, and if no solution for Π_{bb} exists, then Π also must be unsolvable. If the successor generator and goal test are sound, but not necessarily complete, it is still the case that a solution to Π_{bb} is guaranteed to be a solution to Π and hence soundness allows us to reliably use Π_{bb} for producing solutions for Π.

4 Proposed Approach and Methodology

We build upon the previous work that proposed producing a code implementation of $succ$ and $isgoal$ functions [25], taking the human out of the feedback loop. Similar to that work, we care about two properties, *soundness* and *completeness*. As we deal with planning problems described in a natural language, we do not have the formally defined planning task Π. Albeit not stated formally, previous work on generating $succ$ and $isgoal$ with language models assumes the existence of a human expert with the ability to access Π. Examples of such access include a feedback on the code of $succ$ and $isgoal$ produced by the LLM [25] or validating the obtained solution in cases when $succ$ and $isgoal$ are implemented through LLMs [15, 48, 5, 37]. Here, we make a similar assumption, but request a different access to Π. In order to test the soundness and completeness of the produced functions, the human expert is asked for unit tests, information which can provide evidence of unsoundness or incompleteness. The evidence is then used to automatically feedback the model with the information needed to fix the code. We deal with three types of information, exemplified on the 24 Game [48].

1. Examples of inputs to $isgoal$ for which the correct output is known. For instance, we know that $isgoal([24])$ should be true and $isgoal([24, 1])$ should be false.
2. Examples of inputs to $succ$ for which some of the correct outputs are known. For instance, we know that [24], [2], and [-2] are valid successors of [6,4] and therefore should be in $succ([6, 4])$.
3. Partial soundness check for a transition $\langle s, a, t \rangle$ quickly invalidating (obviously) incorrect transitions. For instance, in 24 Game the successor state t must be of length exactly one less than s.

The first two are are usually readily available and often come with the description of the problem. The third one might require some level of understanding of the problem being solved, but it is always possible to use a trivial partial soundness test that always reports that there are no issues. Figure 1 presents an overview of our approach, describing how the provided information is used.

- Step 1 Following Katz et al. (2024), we start with the initial prompts asking for the successor function $succ$ and the goal test $isgoal$.
- Step 2 Then, we perform the goal unit tests, providing feedback to the model in cases of failure, repeatedly asking for a new $isgoal$ until all goal unit tests have passed or a predefined number of iterations was exhausted.

The goal test function failed on the following input state [24, 1], incorrectly reporting it as a goal state. First think step by step what it means for a state to be a goal state in this domain. Then think through in words why the goal test function incorrectly reported input state: [24, 1] as a goal state. Now, revise the goal test function and ensure it returns false for the input state. Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Invalid transformation: length mismatch - the length of a successor must be one less than the parent. Let's think step by step. First think through in words why the successor function produced a successor that had a length that was not exactly one less than the parent. Then provide the complete Python code for the revised successor function that ensures the length of a successor is exactly one less than the parent. Remember how you fixed the previous mistakes, if any. Keep the same function signature.
Input state: [1, 1, 4, 6] Example wrong successor state: [6, 5]

Successor function when run on the state [1, 1, 4, 6] failed to produce all successors. Missing successors are: [[1, 4, 7], [-5, 1, 4], [1, 1, 2], [1, 5, 6], [0.25, 1, 6], [-3, 1, 6], [0.16666666666666666, 1, 4], [1, 3, 6], [1, 4, 5], [1, 1, 1.5]] First think step by step why the successor function failed to produce all successors of the state. Then, fix the successor function. Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Figure 2: **24 Game** example feedback.

Step 3 Once $isgoal$ has passed the unit tests, we perform a soundness check of the current $succ$ and $isgoal$ functions. We do that by plugging these functions in a BFS extended with additional checks and run it on a few example problem instances. If BFS finished, we check whether the goal was indeed reached. If not, that means that $isgoal$ failed to correctly identify a state as a non-goal state and we provide that as feedback to the model, repeating Steps 2 and 3.

Step 4 (Optional) Once the previous steps were finished, we perform the successor unit test, providing feedback to the language model in case of failure.

Every time a goal test fails, we go back to Step 2, every time the successor test fails, we go back to Step 3. After the first step, we always have $succ$ and $isgoal$ that can be plugged into a blind search algorithm. However, if Step 3 fails, we have an indication that we cannot trust the solutions produced by that algorithm. Example feedback produced in Steps 2, 3, and 4 can be seen in Listing 2. In what follows, we provide detailed description of each step of AutoToS.

4.1 System prompt

We instruct the model to provide answers in convenient form for integrating as a search component. Thus, the produced code should consist of a single, self-contained function, as in [51, 47].

Step 1: Initial prompt We take the existing initial prompt from previous work, only ensuring that it includes an example input to the requested function in the correct format [25].

Step 2: Goal function check Goal unit tests assume the existence of a few *known* goal and non-goal states. If the goal function $isgoal$ incorrectly identifies a goal state, then it is incomplete, according to Definition 1. If it incorrectly identifies a non-goal state, then it is not sound. A search with a non-sound goal function can incorrectly report that a solution was found. Whenever an issue with either goal function soundness or completeness was identified, we give feedback to the language model with the description of the failure and the state for which the failure occurred, see example in Figure 2 (top). We use a chain of thought style request, asking the model to discuss why a mistake was made and to generate a fix.

Step 3: Successor function soundness check A soundness check assumes the existence of example problem instances for which we know how to validate that a goal was reached. We extend the BFS/DFS search with additional checks as follows. First, both the successor and goal test functions are wrapped with a timeout of 1 second. These functions should be able to finish in a few milliseconds and therefore 1 second timeout is an indication of an issue with the function. An issue can be as simple as unnecessary computation or multiple successor steps performed instead of a single step or it can even be an infinite loop. Second, the successor function is wrapped with a check whether it modifies the input state. Such modifications often happen when successor states are copied from the input state and modified. Shallow copy of the input state was observed in the previous work [25]. Third, for every successor generated at the expansion step of BFS, a partial soundness check is performed, examining the validity of transitioning from the parent state to the successor state. An example of such a partial soundness check in 24 Game is that the successor state size must be one number less than the parent state. If that does not hold, the successor function is not sound according to Def. 1. It is worth emphasizing that this partial soundness check can be trivial, reporting True for every pair of parent and successor states. If any of the checks did not pass, we provide feedback to the language model with the respective error message, the example input state and the unexpected (or expected and unobserved) output, until all tests are passed or a predefined number of iterations was exhausted (see Fig. 2 middle).

Step 4: Successor function completeness check A successor function completeness check assumes the existence of a few *known* parent and successor states. These can include all successors for some parent state or a subset thereof. If the successor function does not produce some of the known successors, then it is not complete according to Definition 1. While completeness is not required for producing sound solutions, incomplete functions may not generate search space areas where goal states are located, missing existing solutions. Improving completeness is thus an optional step that may improve the accuracy of the produced code. Here as well, we feedback the language model with the respective error message, providing example input state and the missing successors (see Fig. 2).

4.2 Automation, evaluation and validation

Since LLMs are not called during search, there is no need to artificially restrict the algorithms to their incomplete variants [48]; sound and complete algorithms BFS/DFS can be used for solving the search problems. Still, as the human feedback is *before* the feedback loop and the search components produced are not *guaranteed* to be sound, the solutions produced must be validated for soundness.

5 Experiments

To check the feasibility of our approach, AutoToS, we conduct experiments with a representative collection of five search/planning problems: BlocksWorld [14], PrOntoQA [16], Mini Crossword and 24 Game [49], and Sokoban [22]. Four of these domains appeared in ToS [25], while the Sokoban domain did not. We test the performance of various LLMs from three families, using both the largest and smallest models from the same family. Specifically, we use GPT-4o and GPT-4o-Mini [2], Llama3.1-70b and Llama3.1-405b [11], as well as DeepSeek-CoderV2 [10]. We additionally tested Llama3-70b [3], Mistral7x-8b [20], and DeepSeek-CoderV2-Lite, finding these models to perform poorly and therefore excluded from consideration. We use Greedy decoding with maximum context length for each model. For each domain, we restrict the number of calls to the language model per function to 10 (total maximum of 19 per domain). We repeat each experiment 5 times.

Following ToS, we use a simple implementation of BFS and DFS search algorithms in Python. DFS is used for Mini Crosswords, while BFS is used for the other 4 domains. Each successor function execution is limited to 1 second and each overall search is limited to 600 seconds. For each domain, a few (up to 10) instances are used for creating the unit tests. In one case, these instances are taken out of the available set of instances, in other cases we invent new instances. The rest are used for evaluating the accuracy of the generated code, where accuracy measures the percentage of the instances solved. In the case of BFS search, we also require the solution produced to be optimal. This is relevant to BlocksWorld and Sokoban where the solution length matters, but irrelevant for PrOntoQA, where solution is a boolean answer, and 24 Game, where all solutions are of the same length. It is important to emphasize again that if successor function and goal test are sound and complete, then the solution produced by BFS/DFS is guaranteed to be correct (and in the case of BFS optimal). However, since no such guarantees are available, we automatically validate every solution obtained. Experiments were performed on a AMD Ryzen 7 4800H. All models were accessed via API, except for Llama and Deepseek, which were interacted with through a chat interface. Model correspondences logs across all 5 domains are provided in the Appendix.

The aim of our evaluation is to test the following hypotheses. First, whether a partial soundness test improves the accuracy of AutoToS. Second, whether the (optional) completeness step improves the accuracy of AutoToS or not. Third, whether the number of calls to the language model increases significantly compared to ToS. Finally, whether the performance of AutoToS is consistent across different language models of varying sizes.

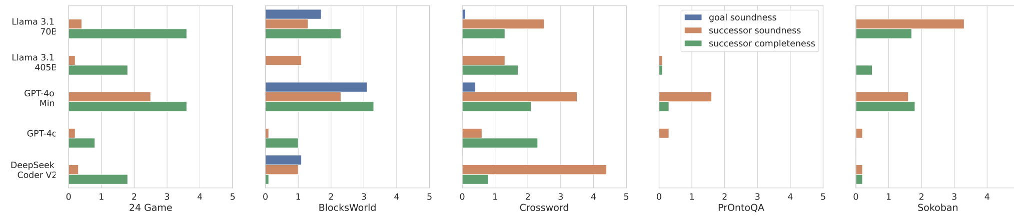

Figure 4: Average # of feedback calls for goal soundness, successor soundness and completeness.

24 Game The 24 Game [49] takes 4 integers in input that can be manipulated through the four arithmetic operations: addition, subtraction, multiplication, and division. The goal is to produce a formula that evaluates to 24, if one exists. States are represented as lists of length 4 or less.

We use the set of 1362 instances [49, 25] and we take out the first 10 instances for unit tests. For the partial soundness test we check whether the number of elements in a successor state is one less than for the parent state. A solution is a sequence of states s_0, s_1, s_2, s_3, where s_0 is the initial state, $s_3 = [24]$ is the goal state, and $\langle s_0, s_1 \rangle$, $\langle s_1, s_2 \rangle$, and $\langle s_2, s_3 \rangle$, are valid transitions. We check that all these hold for a given sequence.

BlocksWorld BlocksWorld is a classic AI planning domain, where the task is to rearrange blocks in towers [14]. There are 4 actions: stack a block on top of another block, unstack a block from another block, put a block down on the table, and pick a block up from the table. States are represented as dictionaries based on 'clear', 'on-table', 'arm-empty', 'holding', and 'on', describing whether a block is clear (no block above it in the tower), the block is on the table, whether the arm is not holding a block and which blocks are on which.

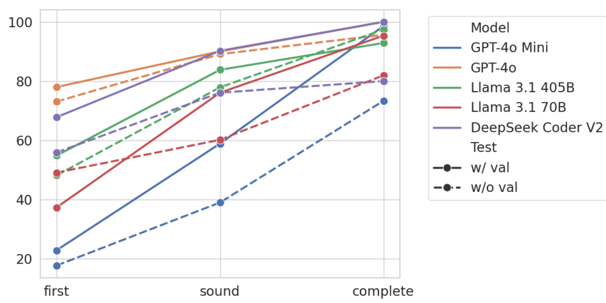

Figure 3: Progression of accuracy values during AutoToS.

The domain has a PDDL representation and a large collection of 502 instances was created by Valmeekam et al. (2023a) and used in the recent literature [15]. We use the entire collection for evaluation and invent 2 example states (and transitions along 2 plans) per unit test. The examples can be found in the Appendix. For the partial soundness test we notice that in each tower there is a top block (that is clear) and there is a bottom block (that is on the table). So, we simply check that the number of blocks in the 'clear' list is the same as in the 'on-table' list. As the instances are given in PDDL, we simply translate the solution into a PDDL format and use an external validator VAL [18].

Mini Crosswords The mini crosswords [49] is a 5x5 crosswords dataset where the input describes the 5 horizontal and 5 vertical clues and the output is the full 25 letters board. We provide a list of horizontal and vertical clues which are strings of words. The verifier ensures that the size of each word in the rows or columns does not exceed 5.

		24 Game	PrOntoQA	Sokoban	Crossword	BlocksWorld
AutoToS	GPT-4o-mini	8.8	4.8	6.4	9.6	10.0
	GPT-4o	3.4	2.6	2.2	5.8	2.0
	Llama3.1-405b	3.4	2.0	2.6	4.0	3.2
	Llama3.1-70b	7.4	2.0	8.2	6.2	5.8
	DeepSeek-CoderV2	4.4	2.0	2.8	6.6	4.2
ToS	GPT-4	2.2	2.6	NA	3.8	3.8

Table 1: Average # of calls to LLM per domain.

We use the existing 20 instances [49, 25], all used for evaluation, with the unit tests constructed based on 3 invented states each, with the successor completeness based on a state in which one horizontal and one vertical clue already filled, which limits the number of possible successors considerably. The partial soundness test verifies that at most 5 new letters are filled in one transition, as well as that the number of unfilled letters does not get larger. A crossword puzzle is solved if the end result is valid, meaning every vertical and horizontal clue is present in the list of possible clues.

PrOntoQA Logical reasoning can be viewed as a search problem of finding a sequence of logical rules that when applied to the known facts, derive or disprove the target hypothesis. Previous work

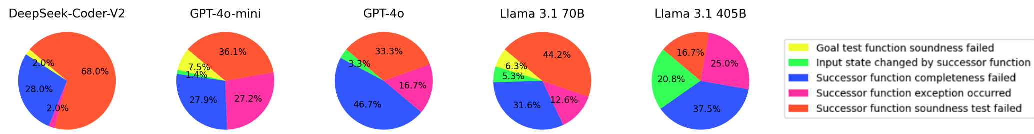

Figure 5: Partition of the errors in the generated code.

applies MCTS with successor function and rewards obtained by calling an LLM, to examples from the PrOntoQA dataset [16] to derive the answer but also the proof, a sequence of reasoning steps. A state is therefore a set of the facts known to be true.

We use the existing set of 4000 instances entirely for evaluation, inventing 3 examples per unit test. A partial soundness test simply checks that each transition adds a single known fact to the state, ensuring that the state size increases by exactly 1. In order to validate the solution, we compare to the known correct answer.

Sokoban Sokoban [22] is a planning problem with PSPACE-complete complexity even for non-optimal planning. The problem, despite its simple conceptual rules, is a notoriously hard for generic AI planners and even for specialized solvers. We use a 2-D grid setup, in which, given equal number of boxes and goal squares, the player needs to push all boxes to goal squares without crossing walls or pushing boxes into walls. The player can only move upward, downward, leftward and rightward where many pushes are irreversible. The domain has a known planning model, described in PDDL of varying grid sizes and difficulties. States are represented as dictionaries with entries: 'at-player,' which represents a single pair of coordinates, and 'at-stone', a list of coordinates for the stones.

We use the collection of PDDL problem instances from the International Planning Competition (IPC) 2008. Out of these instances, we select a subset that can be solved relatively quickly by using the blind search configuration of the efficient planner Fast Downward [17] and choose the instances that were solved in under 5 seconds. This resulted in 11 instances. We use the entire set for evaluation and invent 3 states per unit test. The test simply checks whether the locations of the player and the stones are all different. Similar to BlocksWorld, we translate the solution to PDDL format and use VAL.

Figure 3 depicts the progression of accuracy values for three time points in the process, comparing using the partial soundness test (solid lines, 'w/ val') and not (dotted lines, 'w/o val'). Same colors represent the same language model. The first point in the process corresponds to when the search components are first created, meaning no feedback at all. The second point in the process is when the goal and successor function soundness tests are not failing. The third and final point is the end of the process, when successor completeness tests are not failing. Each point is also annotated with the percentage of cases the step was reached. The aggregation is performed over such cases. The figure allows us to find answers for both the first and the second hypotheses. We can clearly see the benefit from using the partial soundness test, even as simple as the ones we described above. Going forward, we therefore restrict our attention to using the partial soundness test. Further, we can clearly see the strong increase in accuracy when not stopping after the soundness test passes and performing the completeness tests, across all models.

Table 1 shows the total number of calls to the language model until soundness and completeness tests pass. Note that the minimum number of calls is 2, one for each component, even without feedback. We see that the number of automated calls is comparable to the one when a human expert is giving the feedback to the model. To look deeper into how the feedback is partitioned among the three phases, Figure 4 compares the numbers across language models and domains. We see that the larger models rarely require any feedback on the goal function and only a few iterations on the successor function, and more often than not on completeness.

Finally, we can observe that there is no single model that performs better than all other, according to all parameters and the performance is quite consistent across the large models. Interestingly, the smaller model GPT-4o-mini performs quite well in terms of accuracy.

6 Code Errors Discussion

To improve the performance of LLMs in generating search components, it is important to understand the errors in the code produced by these models. We first present the error categories and show the partitioning of the errors to these categories and then elaborate on a few interesting cases.

AutoToS distinguishes 10 error categories and gives each a separate feedback.

1. $succ$ soundness test failed.
2. Input state changed by $succ$.
3. $succ$ completeness failed.
4. $isgoal$ soundness failed.
5. $succ$ exception occurred.
6. $isgoal$ exception occurred.
7. Search timeout in $succ$ soundness test.
8. $succ$ execution took too long.
9. $isgoal$ execution took too long.
10. Response parsing error.

Interestingly, we did not observe any errors in the last two categories. Further only 1, 2, and 3 errors in categories 6, 8, and 7, respectively. The partition of the errors to the other 5 categories (see Figure 5), shows just how much the models differ in the type of errors produced. Interestingly, the DeepSeek-Coder-V2 model rarely produces code that triggers exception or changes the input state and even typically passes the goal soundness test. Other models, especially the smaller ones, are more diverse in errors produced. Across all models, the majority of the errors account for the failed successor soundness and completeness tests.

Bloopers We noticed a few "bloopers," interesting phenomena that occur during AutoToS. We share these observations in a hope of shedding some light onto future understanding of LLM code generation for planning and search problems.

The first blooper occurs in the 5x5 Crossword for Llama3.1-70b. The representation of a Crossword instance includes vertical and horizontal clues which are lists of 5 words each. The model handles horizontal clues well by simply checking whether a word in row i is in the ith list in horizontal clues. For vertical clues, however, the model checks whether the word in column i is at position i among the clues for every column, despite explicit instructions in the prompt.

The second blooper occurs in the GPT-4o-mini, Llama3.1-70b, and even in Llama3.1-405b on the BlocksWorld domain. When generating successors for the unstack block from another block action, the models check if the block is clear, but never actually check whether the arm is empty. The resulting code, in cases when a block is already held, can generate a successor state in which the held block is overwritten with the one that is unstacked, and therefore disappears from the state. On some instances in the evaluation set the situation does not occur. On others, invalid solutions are produced and the accuracy score falls far below 100%. The AutoToS feedback in the next iterations often solves the problem.

Another blooper occurs in Sokoban, when Llama3.1-70b generates the initial successor function and the goal test, and no partial soundness check is performed. The model generates a helper function *is_clear* that only checks whether the location on the grid is 0 or 2 (not a wall), disregarding whether any of the stones are currently at the location. This allows the player to move and push stones to the locations of other stones, resulting in the accuracy score of 0. Since the unit tests pass in this case, no additional iterations were performed.

Yet another blooper happens in 24 Game with GPT-4o-mini and DeepSeek-CoderV2 when no partial soundness check is performed. When creating a new state out of the input state, two numbers are chosen to perform an arithmetic operation and in order to obtain the remaining numbers, the code selects the numbers from the state that are different from the two chosen numbers (compares by value). Thus in cases of duplicate numbers, the state size becomes more than one smaller than of the parent and on some instances the produced solutions would not be valid.

7 Conclusions and Future Work

We automate the process of generating correct and sound code for the search components by leveraging debugging and exception handing with natural language, code feedback, iterative reprompting. We demonstrate the performance of our approach, AutoToS, across various sized models and across search problem domains used by the planning community. With just a few calls to the language model, we demonstrate that we can obtain the search components without any direct human in the loop feedback, ensuring soundness, completeness, accuracy, and nearly 100% accuracy across all models and all domains.

For future work it would be interesting to see if the language models could generate the unit tests as well as the partial soundness tests instead of relying on the user writing these for a specific domain. The partial soundness test is related to the notion of invariants in planning [4]. It is worth exploring whether LLMs can help us derive such invariants. Finally, seeing that smaller language models can achieve accuracy on par with the largest ones, begs the question of whether it would be possible to finetune an even smaller model and achieve a similar or better accuracy.

References

[1] 2023. *Proceedings of the Thirty-Seventh Annual Conference on Neural Information Processing Systems (NeurIPS 2023)*.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[3] AI@Meta. 2024. Llama 3 Model Card. (2024). https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[4] Vidal Alcázar and Álvaro Torralba. 2015. A Reminder about the Importance of Computing and Exploiting Invariants in Planning. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein (Eds.). AAAI Press, 2–6.

[5] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 17682–17690.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021).

[8] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching Large Language Models to Self-Debug. In *ICLR*. OpenReview.net.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. 1990. *Introduction to Algorithms*. The MIT Press.

[10] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, et al. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. arXiv:2405.04434 [cs.CL] https://arxiv.org/abs/2405.04434

[11] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[12] Elliot Gestrin, Marco Kuhlmann, and Jendrik Seipp. 2024. NL2Plan: Robust LLM-Driven Planning from Minimal Text Descriptions. arXiv:2405.04215 [cs.AI] https://arxiv.org/abs/2405.04215

[13] Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. 2023. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems* 36 (2023), 79081–79094.

[14] Naresh Gupta and Dana S. Nau. 1992. On the Complexity of Blocks-World Planning. 56, 2–3 (1992), 223–254.

[15] Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. 2023. Reasoning with Language Model is Planning with World Model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP 2023)*.

[16] Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. 2023. Reasoning with Language Model is Planning with World Model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 8154–8173.

[17] Malte Helmert. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26 (2006), 191–246.

[18] Richard Howey and Derek Long. 2003. VAL's Progress: The Automatic Validation Tool for PDDL2.1 used in the International Planning Competition. In *Proceedings of the ICAPS 2003 Workshop on the Competition: Impact, Organisation, Evaluation, Benchmarks*, Stefan Edelkamp and Jörg Hoffmann (Eds.).

[19] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large Language Models Cannot Self-Correct Reasoning Yet. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=IkmD3fKBPQ

[20] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, et al. 2024. Mixtral of Experts. arXiv:2401.04088 [cs.LG] https://arxiv.org/abs/2401.04088

[21] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A Code Evolution Framework via Large Language Models. arXiv:2306.02907 [cs.CL] https://arxiv.org/abs/2306.02907

[22] Andreas Junghanns and Jonathan Schaeffer. 1997. Sokoban: A Challenging Single-Agent Search Problem. In *International Joint Conference on Artificial Intelligence*. https://api.semanticscholar.org/CorpusID:4049565

[23] Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Paul Saldyt, and Anil B Murthy. 2024. Position: LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. In *Forty-first International Conference on Machine Learning*. https://openreview.net/forum?id=Th8JPEmH4z

[24] Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Paul Saldyt, and Anil B Murthy. 2024. Position: LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. In *Forty-first International Conference on Machine Learning*. https://openreview.net/forum?id=Th8JPEmH4z

[25] Michael Katz, Harsha Kokel, Kavitha Srinivas, and Shirin Sohrabi. 2024. Thought of Search: Planning with Language Models Through The Lens of Efficiency. arXiv:2404.11833 [cs.AI].

[26] Michael Katz, Dany Moshkovich, and Erez Karpas. 2018. Semi-Black Box: Rapid Development of Planning Based Solutions. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*. AAAI Press, 6211–6218.

[27] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

[28] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. 2024. Guiding Enumerative Program Synthesis with Large Language Models. In *CAV (2) (Lecture Notes in Computer Science, Vol. 14682)*. Springer, 280–301.

[29] Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. 2023. LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. *CoRR* abs/2304.11477 (2023).

[30] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 46534–46594. https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf

[31] Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. OctoPack: Instruction Tuning Code Large Language Models. In *ICLR*. OpenReview.net.

[32] James Oswald, Kavitha Srinivas, Harsha Kokel, Junkyu Lee, Michael Katz, and Shirin Sohrabi. 2024. Large Language Models as Planning Domain Generators. In *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)*, Sara Bernardini and Christian Muise (Eds.). AAAI Press.

[33] Vishal Pallagani, Bharath Muppasani, Keerthiram Murugesan, Francesca Rossi, Lior Horesh, Biplav Srivastava, Francesco Fabiano, and Andrea Loreggia. 2022. Plansformer: Generating Symbolic Plans using Transformers. arXiv:2212.08681 [cs.AI].

[34] Vishal Pallagani, Bharath Muppasani, Keerthiram Murugesan, Francesca Rossi, Biplav Srivastava, Lior Horesh, Francesco Fabiano, and Andrea Loreggia. 2023. Understanding the Capabilities of Large Language Models for Automated Planning. *CoRR* abs/2305.16151 (2023).

[35] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *NeurIPS Datasets and Benchmarks*.

[36] Shreyas Sundara Raman, Vanya Cohen, Eric Rosen, Ifrah Idrees, David Paulius, and Stefanie Tellex. 2022. Planning With Large Language Models Via Corrective Re-Prompting. In *NeurIPS 2022 Foundation Models for Decision Making Workshop*. https://openreview.net/forum?id=cMDMRBe1TKs

[37] Bilgehan Sel, Ahmad Al-Tawaha, Vanshaj Khattar, Lu Wang, Ruoxi Jia, and Ming Jin. 2023. Algorithm of Thoughts: Enhancing Exploration of Ideas in Large Language Models. *CoRR* abs/2308.10379 (2023).

[38] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning, See [1].

[39] Tom Silver, Soham Dan, Kavitha Srinivas, Josh Tenenbaum, Leslie Pack Kaelbling, and Michael Katz. 2024. Generalized Planning in PDDL Domains with Pretrained Large Language Models. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2024)*, Jennifer Dy and Sriraam Natarajan (Eds.). AAAI Press.

[40] Tom Silver, Varun Hariprasad, Reece S Shuttleworth, Nishanth Kumar, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. 2022. PDDL Planning with Pretrained Large Language Models. In *NeurIPS 2022 Foundation Models for Decision Making Workshop*. https://openreview.net/forum?id=1QMMUB4zfl

[41] Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. 2024. On the Self-Verification Limitations of Large Language Models on Reasoning and Planning Tasks. *arXiv preprint arXiv:2402.08115* (2024).

[42] Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. 2023. PlanBench: An Extensible Benchmark for Evaluating Large Language Models on Planning and Reasoning about Change, See [1], 38975–38987.

[43] Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. 2023. On the Planning Abilities of Large Language Models - A Critical Investigation, See [1].

[44] Zihao Wang, Anji Liu, Haowei Lin, Jiaqi Li, Xiaojian Ma, and Yitao Liang. 2024. Rat: Retrieval augmented thoughts elicit context-aware reasoning in long-horizon generation. *arXiv preprint arXiv:2403.05313* (2024).

[45] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[46] Yaqi Xie, Chen Yu, Tongyao Zhu, Jinbin Bai, Ze Gong, and Harold Soh. 2023. Translating Natural Language to Planning Goals with Large-Language Models. *CoRR* abs/2302.05128 (2023).

[47] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793* (2024).

[48] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models, See [1].

[49] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Thirty-seventh Conference on Neural Information Processing Systems*. `https://openreview.net/forum?id=5Xc1ecxO1h`

[50] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. In *ACL (1)*. Association for Computational Linguistics, 769–787.

[51] Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step-by-step. arXiv:2402.16906 [cs.SE] `https://arxiv.org/abs/2402.16906`

[52] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language Agent Tree Search Unifies Reasoning Acting and Planning in Language Models. *CoRR* abs/2310.04406 (2023).

[53] Max Zuo, Francisco Piedrahita Velez, Xiaochen Li, Michael L. Littman, and Stephen H. Bach. 2024. Planetarium: A Rigorous Benchmark for Translating Text to Structured Planning Languages. *CoRR* abs/2407.03321 (2024).

8 Additional data for experimental domains

We provide additional information on the domains included in our experimental evaluation, such as examples used in unit tests, code for the partial successor soundness test, etc.

8.1 24 Game

Goal Unit Test Goal unit test cases are stored in two *jsonl* files, one for goal states and one for non-goal states.

[h] [frame=single, framesep=3mm, linenos=true, xleftmargin=21pt, tabsize=4]js [24]

[h] [frame=single, framesep=3mm, linenos=true, xleftmargin=21pt, tabsize=4]js [] [3] [24, 1] [1, 6, 4] [1, 1, 4, 6]

Successor Unit Test Successor unit test cases are stored in a jsonl file. The test cases used are depicted in Listing 8.1.

[frame=single, framesep=3mm, linenos=true, xleftmargin=21pt,breaklines, tabsize=4]js [[1, 1, 4, 6], [[1, 1, 10], [0.6666666666666666, 1, 1], [1, 4, 7], [-2, 1, 1], [-5, 1, 4], [1, 4, 6], [1, 1, 2], [1, 5, 6], [0.25, 1, 6], [-3, 1, 6], [0, 4, 6], [0.16666666666666666, 1, 4], [1, 1, 24], [1, 3, 6], [2, 4, 6], [1, 4, 5], [1, 1, 1.5]]] [[1, 1, 11, 11], [[1, 11, 11], [0.09090909090909091, 1, 11], [0, 11, 11], [1, 1, 22], [2, 11, 11], [0, 1, 1], [1, 1, 121], [1, 11, 12], [1, 1, 1.0], [1, 10, 11], [-10, 1, 11]]] [[1, 1, 3, 8], [[-2, 1, 8], [1, 3, 8], [0.3333333333333333, 1, 8], [-7, 1, 3], [1, 1, 2.6666666666666665], [0.125, 1, 3], [2, 3, 8], [1, 3, 7], [1, 1, 11], [1, 1, 5], [1, 1, 24], [-5, 1, 1], [0.375, 1, 1], [1, 2, 8], [1, 3, 9], [0, 3, 8], [1, 4, 8]]] [[1, 1, 1, 8], [[0.125, 1, 1], [1, 1, 9], [1, 1, 8], [0, 1, 8], [1, 2, 8], [1, 1, 7], [-7, 1, 1]]] [[6, 6, 6, 6], [[1.0, 6, 6], [6, 6, 12], [0, 6, 6], [6, 6, 36]]] [[1, 1, 2, 12], [[1, 3, 12], [-10, 1, 1], [1, 1, 10], [2, 2, 12], [1, 2, 13], [0.5, 1, 12], [-11, 1, 2], [1, 1, 12], [1, 1, 6.0], [1, 2, 12], [0, 2, 12], [1, 1, 24], [1, 2, 11], [1, 1, 14], [0.16666666666666666, 1, 1], [0.08333333333333333, 1, 2], [-1, 1, 12]]] [[1, 2, 2, 6], [[2, 2, 6], [-5, 2, 2], [2, 3, 6], [1, 2, 6], [0.3333333333333333, 1, 2], [2, 2, 5], [1, 1.0, 6], [0.16666666666666666, 2, 2], [1, 4, 6], [0, 1, 6], [-4, 1, 2], [1, 2, 12], [1, 2, 3.0], [2, 2, 7], [-1, 2, 6], [1, 2, 8], [1, 2, 4], [0.5, 2, 6]]] [[1, 1, 10, 12], [[-9, 1, 12], [1, 1, 1.2], [0.08333333333333333, 1, 10], [-2, 1, 1], [1, 10, 13], [1, 1, 22], [2, 10, 12], [0.1, 1, 12], [1, 1, 120], [1, 1, 2], [0.8333333333333334, 1, 1], [1, 9, 12], [1, 10, 12], [0, 10, 12], [1, 11, 12], [1, 10, 11], [-11, 1, 10]]] [[2, 2, 10, 10], [[0, 2, 2], [2, 10, 12], [1.0, 10, 10], [0, 10, 10], [-8, 2, 10], [2, 2, 100], [2, 5.0, 10], [1.0, 2, 2], [2, 2, 20], [4, 10, 10], [0.2, 2, 10], [2, 8, 10], [2, 10, 20]]] [[1, 1, 1, 12], [[0.08333333333333333, 1, 1], [1, 1, 13], [1, 1, 12], [0, 1, 12], [1, 2, 12], [-11, 1, 1], [1, 1, 11]]] [[1, 4, 6], [[4, 6]]] [[4, 6], [[24]]] [[1, 1, 22], [[1, 23]]] [[1, 23], [[24]]] [[1, 1, 24], [[1, 24]]] [[1, 24], [[24]]] [[1, 2, 8], [[3, 8]]] [[3, 8], [[24]]] [[6, 6, 12], [[6, 18]]] [[6, 18], [[24]]] [[0, 2, 12], [[0, 24]]] [[0, 24], [[24]]] [[2, 2, 6], [[2, 12]]] [[2, 12], [[24]]] [[1, 11, 12], [[11, 13]]] [[11, 13], [[24]]] [[2, 2, 20], [[2, 22]]] [[2, 22], [[24]]] [[1, 1, 12], [[2, 12]]]

Partial Successor Soundness Test The code for the partial successor soundness test is as follows.

[breaklines]python def validate$_{t}ransition_{c}omplex(s, t) : iflen(s) - len(t)! = 1 : feedback = prettyprint(''Invalidtransformation : lengthmismatch - thelengthofasuccessormustbeonelessthantheparent.'')feedback+ = prettyprint(''Let'sthinkstepbystep.Firstthinkthroughinwordswhythesuccessorfunctionproducedasuccessorthatha$ $prettyprint(''Rememberhowyoufixedthepreviousmistakes, ifany.Keepthesamefunctionsignature.'')returnFalse,$

8.2 Blocksworld

Goal Unit Test Goal unit test cases are stored in two *jsonl* files, one for goal states and one for non-goal states, depicted in Listings 8.2 and 8.2.

[h] [frame=single, framesep=3mm, linenos=true, breaklines, xleftmargin=21pt, tabsize=4]js ["state": "clear": ["b"], "on-table": ["d"], "arm-empty": true, "holding": null, "on": [["a", "c"],["b", "a"],["c","d"]] , "goal": "clear": [], "on-table": [], "on": [["a","c"],["b","a"],["c","d"]] , "state": "clear": ["a"], "on-table": ["d"], "arm-empty": false, "holding": "b", "on": [["a","c"],["c","d"]] , "goal": "clear": [], "on-table": [], "on": [["a","c"]]]

[h] [frame=single, framesep=3mm, linenos=true, breaklines, xleftmargin=21pt, tabsize=4]js ["state": "clear": ["b"], "on-table": ["d"], "arm-empty": true, "holding": null, "on": [["a","c"],["b","a"],["c","d"]] , "goal": "clear": [], "on-table": [], "on": [["a","b"],["b","c"],["c","d"]] , "state": "clear": ["a"], "on-table": ["d"], "arm-empty": false, "holding": "b", "on": [["a","c"],["c","d"]] , "goal": "clear": [], "on-table": [], "on": [["a","c"],["c","b"]]]

Successor Unit Test Successor unit test cases are stored in a jsonl file, depicted in Listing 8.2.

[frame=single, framesep=3mm, linenos=true, breaklines, xleftmargin=21pt, tabsize=4]js ["state": "clear": ["b"], "on-table": ["d"], "arm-empty": true, "holding": null, "on": [["a","c"],["b","a"],["c","d"]] , "successors": ["clear": ["a"], "on-table": ["d"], "arm-empty": false, "holding":

"b", "on": [["a","c"],["c","d"]]] , "state": "clear": ["a"], "on-table": ["d"], "arm-empty": false, "holding": "b", "on": [["a","c"],["c","d"]] , "successors": ["clear": ["a","b"], "on-table": ["d","b"], "arm-empty": true, "holding": null, "on": [["a","c"],["c","d"]] , "clear": ["b"], "on-table": ["d"], "arm-empty": true, "holding": null, "on": [["a","c"],["c","d"],["b","a"]]] , "state": "clear": ["a","b","d"], "on-table": ["a","c","d"], "arm-empty": true, "holding": null, "on": [["b","c"]] , "successors": ["clear": ["b","d"], "on-table": ["c","d"], "arm-empty": false, "holding": "a", "on": [["b","c"]] , "clear": ["a","b"], "on-table": ["a","c"], "arm-empty": false, "holding": "d", "on": [["b","c"]] , "clear": ["a","d","c"], "on-table": ["a","c","d"], "arm-empty": false, "holding": "b", "on": []] , "state": "clear": ["b","d"], "on-table": ["c","d"], "arm-empty": false, "holding": "a", "on": [["b","c"]] , "successors": ["clear": ["b","d","a"], "on-table": ["c","d","a"], "arm-empty": true, "holding": null, "on": [["b","c"]] , "clear": ["d","a"], "on-table": ["c","d"], "arm-empty": true, "holding": null, "on": [["b","c"],["a","b"]] , "clear": ["b","a"], "on-table": ["c","d"], "arm-empty": true, "holding": null, "on": [["b","c"],["a","d"]]] , "state": "clear": ["a","d"], "on-table": ["b","c"], "arm-empty": true, "holding": null, "on": [["a","b"],["d","c"]] , "successors": ["clear": ["d","b"], "on-table": ["b","c"], "arm-empty": false, "holding": "a", "on": [["d","c"]] , "clear": ["a","c"], "on-table": ["b","c"], "arm-empty": false, "holding": "d", "on": [["a","b"]]] , "state": "clear": ["d","b"], "on-table": ["b","c"], "arm-empty": false, "holding": "a", "on": [["d","c"]] , "successors": ["clear": ["d","b","a"], "on-table": ["b","c","a"], "arm-empty": true, "holding": null, "on": [["d","c"]] , "clear": ["b","a"], "on-table": ["b","c"], "arm-empty": true, "holding": null, "on": [["d","c"],["a","d"]] , "clear": ["d","a"], "on-table": ["b","c"], "arm-empty": true, "holding": null, "on": [["d","c"],["a","b"]]] , "state": "clear": ["b"], "on-table": ["a"], "arm-empty": true, "holding": null, "on": [["b","c"],["c","d"],["d","a"]] , "successors": ["clear": ["c"], "on-table": ["a"], "arm-empty": false, "holding": "b", "on": [["c","d"],["d","a"]]] , "state": "clear": ["c"], "on-table": ["a"], "arm-empty": false, "holding": "b", "on": [["c","d"],["d","a"]] , "successors": ["clear": ["c","b"], "on-table": ["a","b"], "arm-empty": true, "holding": null, "on": [["c","d"],["d","a"]] , "clear": ["b"], "on-table": ["a"], "arm-empty": true, "holding": null, "on": [["c","d"],["d","a"],["b","c"]]] , "state": "clear": ["d"], "on-table": ["b"], "arm-empty": true, "holding": null, "on": [["a","c"],["c","b"],["d","a"]] , "successors": ["clear": ["a"], "on-table": ["b"], "arm-empty": false, "holding": "d", "on": [["a","c"],["c","b"]]] , "state": "clear": ["a"], "on-table": ["b"], "arm-empty": false, "holding": "d", "on": [["a","c"],["c","b"]] , "successors": ["clear": ["a","d"], "on-table": ["b","d"], "arm-empty": true, "holding": null, "on": [["a","c"],["c","b"]] , "clear": ["d"], "on-table": ["b"], "arm-empty": true, "holding": null, "on": [["a","c"],["c","b"],["d","a"]]] , "state": "clear": ["c","d"], "on-table": ["a","d"], "arm-empty": true, "holding": null, "on": [["b","a"],["c","b"]] , "successors": ["clear": ["c"], "on-table": ["a"], "arm-empty": false, "holding": "d", "on": [["b","a"],["c","b"]] , "clear": ["d","b"], "on-table": ["a","d"], "arm-empty": false, "holding": "c", "on": [["b","a"]]] , "state": "clear": ["c"], "on-table": ["a"], "arm-empty": false, "holding": "d", "on": [["b","a"],["c","b"]] , "successors": ["clear": ["c","d"], "on-table": ["a","d"], "arm-empty": true, "holding": null, "on": [["b","a"],["c","b"]] , "clear": ["d"], "on-table": ["a"], "arm-empty": true, "holding": null, "on": [["b","a"],["c","b"],["d","c"]]] , "state": "clear": ["d"], "on-table": ["b"], "arm-empty": true, "holding": null, "on": [["a","c"],["c","b"],["d","a"]] , "successors": ["clear": ["a"], "on-table": ["b"], "arm-empty": false, "holding": "d", "on": [["a","c"],["c","b"]]] , "state": "clear": ["a"], "on-table": ["b"], "arm-empty": false, "holding": "d", "on": [["a","c"],["c","b"]] , "successors": ["clear": ["a","d"], "on-table": ["b","d"], "arm-empty": true, "holding": null, "on": [["a","c"],["c","b"]] , "clear": ["d"], "on-table": ["b"], "arm-empty": true, "holding": null, "on": [["a","c"],["c","b"],["d","a"]]] , "state": "clear": ["a"], "on-table": ["c"], "arm-empty": true, "holding": null, "on": [["a","d"],["b","c"],["d","b"]] , "successors": ["clear": ["d"], "on-table": ["c"], "arm-empty": false, "holding": "a", "on": [["b","c"],["d","b"]]] , "state": "clear": ["d"], "on-table": ["c"], "arm-empty": false, "holding": "a", "on": [["b","c"],["d","b"]] , "successors": ["clear": ["d","a"], "on-table": ["c","a"], "arm-empty": true, "holding": null, "on": [["b","c"],["d","b"]] , "clear": ["a"], "on-table": ["c"], "arm-empty": true, "holding": null, "on": [["b","c"],["d","b"],["a","d"]]] , "state": "clear": ["a","b","d"], "on-table": ["a","c","d"], "arm-empty": true, "holding": null, "on": [["b","c"]] , "successors": ["clear": ["b","d"], "on-table": ["c","d"], "arm-empty": false, "holding": "a", "on": [["b","c"]] , "clear": ["a","b"], "on-table": ["a","c"], "arm-empty": false, "holding": "d", "on": [["b","c"]] , "clear": ["a","d","c"], "on-table": ["a","c","d"], "arm-empty": false, "holding": "b", "on": []] , "state": "clear": ["b","d"], "on-table": ["c","d"], "arm-empty": false, "holding": "a", "on": [["b","c"]] , "successors": ["clear": ["b","d","a"], "on-table": ["c","d","a"], "arm-empty": true, "holding": null, "on": [["b","c"]] , "clear": ["d","a"], "on-table": ["c","d"], "arm-empty": true, "holding": null, "on": [["b","c"],["a","b"]] , "clear": ["b","a"], "on-table": ["c","d"], "arm-empty": true, "holding": null, "on": [["b","c"],["a","d"]]] , "state": "clear": ["b","c"], "on-table": ["a","b"], "arm-empty": true, "holding": null, "on": [["c","d"],["d","a"]] , "successors": ["clear": ["c"], "on-table": ["a"], "arm-empty": false, "holding": "b", "on": [["c","d"],["d","a"]] , "clear": ["b","d"], "on-table": ["a","b"], "arm-empty": false, "holding": "c", "on": [["d","a"]]] , "state": "clear": ["c"], "on-table": ["a"], "arm-empty": false, "holding": "b", "on": [["c","d"],["d","a"]] , "successors": ["clear": ["c","b"], "on-table": ["a","b"], "arm-empty": true, "holding": null, "on": [["c","d"],["d","a"]] , "clear": ["b"], "on-table": ["a"], "arm-empty": true, "holding": null, "on": [["c","d"],["d","a"],["b","c"]]]]

Partial Successor Soundness Test [breaklines]python def validate$_transition_complex(parent, state)$: $iflen(state.get('clear'))! = len(state.get('on - table'))$: $feedback+ = prettyprint("Eachtowerhasthebottomblockonthetableandthetopblockclear.")feedback+ = prettyprint("Therefore, thenumberofclearblocksshouldbethesameasthenumberofblocksonthetable.")feedback+ = prettyprint("Thenumberofelementsintheclearlistisnotthesameasthenumberofelementsintheon - tablelist.")feedback+ = prettyprint("Reminder : OnceIpickupablock, Iamholdingtheblockanditisnolongerclearandnolongeronthetable.")fee$ $prettyprint("OnceIunstackfromontopofanotherblock, Iamholdingtheblockanditisnolongerclear.Instead, theotherblockbecomesclear.")feedb$ $prettyprint("OnceIputdownablock, myhandbecomesempty, theblockbecomesclear, anditisnowonthetable.")feedback+ = prettyprint("OnceIstackablockontopofanotherblock, theblockontopbecomesclearandtheblockunderitisnolongerclear.")$

feedback += prettyprint("Let's think step by step. First, think of how applying each action changes which blocks are clear.") feedback += prettyprint("Then, think of how applying each action changes which blocks are on the table.") feedback += prettyprint("Then, provide the complete Python code for the revised successor function that returns a list of successor states.") feedback += prettyprint("Remember how you fixed the previous mistakes, if any. Keep the same function signature.") return False, feedback return True, ""

8.3 5x5 Crosswords

Goal Unit Test Goal unit test cases are stored in two *jsonl* files, one for goal states and one for non-goal states.

[frame=single, framesep=3mm, linenos=true, xleftmargin=21pt, breaklines, tabsize=4]js ["state": [["a", "g", "e", "n", "d"], ["m", "o", "t", "o", "r"], ["a", "r", "t", "s", "y"], ["s", "a", "l", "l", "e"], ["s", "l", "e", "e", "r"]], "horizontal$_{c}lues"$: [[$"tasks", "goals", "plans", "agend", "chores", "works", "deeds", "items", "lists", "brief"], ["motor", "power", "drive", "diesel", "steam", "pur$ [[$"amass", "stack", "hoard", "pile", "store", "heaps", "massy", "gathe", "lumps", "mound"], ["nilga", "goral", "eland", "lepus", "gazal", "kudi$

[frame=single, framesep=3mm, linenos=true, xleftmargin=21pt, breaklines, tabsize=4]js ["state": [[null, null, null, null, null], ["m", "o", "t", "o", "r"], ["a", "r", "t", "s", "y"], ["s", "a", "l", "l", "e"], ["s", "l", "e", "e", "r"]], "horizontal$_{c}lues"$: [[$"tasks", "goals", "plans", "agend", "chores", "works", "deeds", "items", "lists", "brief"], ["motor", "power", "drive", "diesel", "steam", "pur$ [[$"amass", "stack", "hoard", "pile", "store", "heaps", "massy", "gathe", "lumps", "mound"], ["nilga", "goral", "eland", "lepus", "gazal", "kudi$

Successor Unit Test Successor unit test cases are stored in a jsonl file. The test cases used are depicted in Listing 8.3.

[frame=single, framesep=3mm, linenos=true, xleftmargin=21pt,breaklines, tabsize=4]js ["state": [[null, null, "e", null, null], ["m", "o", "t", "o", "r"], [null, null, "t", null, null], [null, null, "l", null, null], [null, null, "e", null, null]], "successors": [[["a", "g", "e", "n", "d"], ["m", "o", "t", "o", "r"], [null, null, "t", null, null], [null, null, "l", null, null], [null, null, "e", null, null]], [["d", "e", "e", "d", "s"], ["m", "o", "t", "o", "r"], [null, null, "t", null, null], [null, null, "l", null, null], [null, null, "e", null, null]], [["i", "t", "e", "m", "s"], ["m", "o", "t", "o", "r"], [null, null, "t", null, null], [null, null, "l", null, null], [null, null, "e", null, null]], [[null, null, "e", null, null], ["m", "o", "t", "o", "r"], ["a", "r", "t", "s", "y"], [null, null, "l", null, null], [null, null, "e", null, null]], [[null, null, "e", null, null], ["m", "o", "t", "o", "r"], [null, null, "t", null, null], ["s", "a", "l", "l", "e"], [null, null, "e", null, null]], [[null, null, "e", null, null], ["m", "o", "t", "o", "r"], [null, null, "t", null, null], ["m", "a", "l", "l", "s"], [null, null, "e", null, null]], [[null, null, "e", null, null], ["m", "o", "t", "o", "r"], [null, null, "t", null, null], [null, null, "l", null, null], ["s", "l", "e", "e", "r"]], [[null, null, "e", null, null], ["m", "o", "t", "o", "r"], [null, null, "t", null, null], [null, null, "l", null, null], ["s", "n", "e", "e", "r"]], [["a", null, "e", null, null], ["m", "o", "t", "o", "r"], ["a", null, "t", null, null], ["s", null, "l", null, null], ["s", null, "e", null, null]], [[null, "g", "e", null, null], ["m", "o", "t", "o", "r"], [null, "r", "t", null, null], [null, "a", "l", null, null], [null, "l", "e", null, null]], [[null, null, "e", "n", null], ["m", "o", "t", "o", "r"], [null, null, "t", "s", null], [null, null, "l", "l", null], [null, null, "e", "e", null]], [[null, null, "e", "m", null], ["m", "o", "t", "o", "r"], [null, null, "t", "u", null], [null, null, "l", "t", null], [null, null, "e", "h", null]], [[null, null, "e", "n", null], ["m", "o", "t", "o", "r"], [null, null, "t", "s", null], [null, null, "l", "t", null], [null, null, "e", "r", null]], [[null, null, "e", "p", null], ["m", "o", "t", "o", "r"], [null, null, "t", "r", null], [null, null, "l", "t", null], [null, null, "e", "s", null]], [[null, null, "e", null, "d"], ["m", "o", "t", "o", "r"], [null, null, "t", null, "i"], [null, null, "l", null, "e"], [null, null, "e", null, "r"]], [[null, null, "e", null, "d"], ["m", "o", "t", "o", "r"], [null, null, "t", null, "y"], [null, null, "l", null, "e"], [null, null, "e", null, "r"]], [[null, null, "e", null, "w"], ["m", "o", "t", "o", "r"], [null, null, "t", null, "i"], [null, null, "l", null, "n"], [null, null, "e", null, "g"]], [[null, null, "e", null, "d"], ["m", "o", "t", "o", "r"], [null, null, "t", null, "e"], [null, null, "l", null, "a"], [null, null, "e", null, "r"]]], "horizontal$_{c}lues"$: [[$"tasks", "goals", "plans", "agend", "chores", "works", "deeds", "items", "lists", "brief"], ["motor", "power", "drive", "diesel", "steam", "pur$ [[$"amass", "stack", "hoard", "pile", "store", "heaps", "massy", "gathe", "lumps", "mound"], ["nilga", "goral", "eland", "lepus", "gazal", "kudi$

Partial Successor Soundness Test [breaklines]python def validate$_{t}ransition_{c}omplex(s, t) : def count_{n}one(s) : ns = 0 for r in s : ns += len([c for c in r if c is None]) return ns$

ns = count$_{n}one(s) nt = count_{n}one(t)$

feedback = "" if ns < nt: More unknown feedback += prettyprint("Successor state has less filled cells than the parent state.") elif ns == nt: Same unknown feedback += prettyprint("Successor state has the same number of filled cells as the parent state.") elif ns - nt > 5: Way too many less unknown feedback += prettyprint("Successor state has more than 5 filled cells more than the parent state.") else: return True, ""

feedback += prettyprint("Let's think step by step. First, think what you did wrong.") feedback += prettyprint("Then, think of in what ways successor state should be different from the parent state.") feedback += prettyprint("Then, provide the complete Python code for the revised successor function that returns a list of successor states.") feedback += prettyprint("Remember how you fixed the previous mistakes, if any. Keep the same function signature.") return False, feedback

8.4 ProntoQA

Goal Unit Test Goal unit test cases are stored in two *jsonl* files, one for goal states and one for non-goal states.

[h] [frame=single, framesep=3mm, linenos=true, xleftmargin=21pt, tabsize=4]js "state": ["painted lady", "bony"], "goal": "bony" "state": ["mersenne prime", "real"], "goal": "real" "state": ["lepidopteran", "small"], "goal": "small"

[h] [frame=single, framesep=3mm, linenos=true, xleftmargin=21pt, tabsize=4]js "state": ["painted lady"], "goal": "not-bony" "state": ["mersenne prime"], "goal": "not-real" "state": ["lepidopteran"], "goal": "not-small"

Successor Unit Test Successor unit test cases are stored in a jsonl file. The test cases used are depicted in Listing 8.4.

[frame=single, framesep=3mm, linenos=true, xleftmargin=21pt,breaklines, tabsize=4]js "state": ["painted lady"], "rules": [["arthropod", "protostome"], ["lepidopteran", "insect"], ["painted lady", "butterfly"], ["insect", "arthropod"], ["invertebrate", "animal"], ["arthropod", "not-bony"], ["protostome", "invertebrate"], ["whale", "bony"], ["butterfly", "lepidopteran"], ["animal", "multicellular"], ["insect", "six-legged"]], "successors": [["painted lady", "butterfly"]] "state": ["mersenne prime"], "rules": [["integer", "real number"], ["prime number", "natural number"], ["real number", "number"], ["mersenne prime", "prime number"], ["mersenne prime", "not-composite"], ["natural number", "integer"], ["imaginary number", "not-real"], ["real number", "real"], ["prime number", "not-composite"], ["natural number", "positive"]], "successors": [["prime number", "mersenne prime"], ["not-composite", "mersenne prime"]] "state": ["lepidopteran"], "rules": [["lepidopteran", "insect"], ["arthropod", "small"], ["insect", "arthropod"], ["whale", "not-small"], ["invertebrate", "animal"], ["butterfly", "lepidopteran"], ["arthropod", "invertebrate"], ["animal", "multicellular"], ["insect", "six-legged"]], "successors": [["insect", "lepidopteran"]]

Partial Successor Soundness Test [breaklines]python def validate$_{t}ransition_{c}omplex(s,t)$: $ifs == t : returnTrue,""eliflen(t) - len(s)! = 1 : feedback = prettyprint("Invalidtransition : lengthmismatch - thelengthofasuccessormustbeonemorethantheparent.")feedback+ = prettyprint("Let'sthinkstepbystep.Firstthinkthroughinwordswhythesuccessorfunctionproducedasuccessorthatha$ $prettyprint("Rememberhowyoufixedthepreviousmistakes, ifany.Keepthesamefunctionsignature.")returnFalse,$

8.5 Sokoban

Goal Unit Test Goal unit test cases are stored in two *jsonl* files, one for goal states and one for non-goal states.

[h] [frame=single, framesep=3mm, linenos=true, xleftmargin=21pt, breaklines, tabsize=4]js "state": "at-player": [5, 3], "at-stone": [[3, 3], [4, 3]], "grid": [[1, 1, 1, 1, 1, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 2, 0, 1], [1, 0, 1, 2, 1, 1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]] "state": "at-player": [5, 2], "at-stone": [[3, 2], [4, 2]], "grid": [[1, 0, 1, 1, 1, 1, 1], [0, 0, 1, 0, 0, 0, 1], [1, 1, 1, 0, 0, 0, 1], [1, 0, 2, 0, 0, 0, 1], [1, 0, 2, 1, 0, 0, 1], [1, 0, 0, 1, 0, 0, 1], [1, 1, 1, 1, 1, 1, 1]] "state": "at-player": [4, 4], "at-stone": [[2, 2], [3, 3]], "grid": [[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 2, 0, 1, 1, 1, 1], [1, 0, 0, 2, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0, 1, 1, 1, 0, 0, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]

[h] [frame=single, framesep=3mm, linenos=true, xleftmargin=21pt, breaklines, tabsize=4]js "state": "at-player": [5, 3], "at-stone": [[3, 3], [4, 3]], "grid": [[1, 1, 1, 1, 1, 1], [1, 0, 0, 2, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 2, 1, 1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]] "state": "at-player": [5, 2], "at-stone": [[3, 2], [4, 2]], "grid": [[1, 0, 1, 1, 1, 1, 1], [0, 0, 1, 0, 0, 0, 1], [1, 1, 1, 0, 0, 2, 1], [1, 0, 0, 0, 0, 0, 1], [1, 0, 0, 1, 0, 2, 1], [1, 0, 0, 1, 0, 0, 1], [1, 1, 1, 1, 1, 1, 1]] "state": "at-player": [4, 4], "at-stone": [[2, 2], [3, 3]], "grid": [[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1, 1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0, 1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]

Successor Unit Test Successor unit test cases are stored in a jsonl file. The test cases used are depicted in Listing 8.5.

[frame=single, framesep=3mm, linenos=true, xleftmargin=21pt,breaklines, tabsize=4]js "state": "at-player": [5, 3], "at-stone": [[3, 3], [4, 3]], "successors": ["at-player": [5, 2], "at-stone": [[3, 3], [4, 3]]], "grid": [[1, 1, 1, 1, 1, 1], [1, 0, 0, 2, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 2, 1, 1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]] "state": "at-player": [5, 2], "at-stone": [[3, 2], [4, 2]], "successors": ["at-player": [5, 1], "at-stone": [[3, 2], [4, 2]]], "grid": [[1, 0, 1, 1, 1, 1, 1], [0, 0, 1, 0, 0, 0, 1], [1, 1, 1, 0, 0, 2, 1], [1, 0, 0, 0, 0, 0, 1], [1, 0, 0, 1, 0, 2, 1], [1, 0, 0, 1, 0, 0, 1], [1, 1, 1, 1, 1, 1, 1]] "state": "at-player": [4, 4], "at-stone": [[2, 2], [3, 3]], "successors": ["at-player": [5, 4], "at-stone": [[2, 2], [3, 3]], "at-player": [4, 3], "at-stone": [[2, 2], [3, 3]], "at-player": [4, 5], "at-stone": [[2, 2], [3, 3]]], "grid": [[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1, 1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0, 1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 1, 1, 1]] "state": "at-player": [5, 3], "at-stone": [[5, 2], [4, 3]], "successors": ["at-player": [5, 2], "at-stone": [[5, 1], [4, 3]]], "grid": [[1, 1, 1, 1, 1, 1], [1, 0, 0, 2, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 2, 1, 1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]]

Partial Successor Soundness Test [breaklines]python def validate$_{t}ransition_{c}omplex(s,t)$: $locations = set(t['at - stone'])iflen(locations) < len(t['at - stone']) : feedback = prettyprint("Invalidtransition : multiplestonesatthesamelocation.")feedback+ = prettyprint("Let'sthinkstepbystep.Firstthinkthroughinwordswhythesuccessorfunctionproducedasuccessorthatha$ $prettyprint("Rememberhowyoufixedthepreviousmistakes, ifany.Keepthesamefunctionsignature.")returnFalse,$ $player']inlocations : feedback = prettyprint("Invalidtransition :$

$astoneandtheplayerareatthesamelocation.")feedback+=prettyprint("Let'sthinkstepbystep.Firstthinkthroughin$
$prettyprint("Rememberhowyoufixedthepreviousmistakes,ifany.Keepthesamefunctionsignature.")returnFalse,$

9 Model Correspondences

We show one full correspondence with the language model per domain, LLM, and with or without validation. The correspondences include *AutoToS prompt*, *Model response*, as well as *System messages* that log the current state of the system.

9.1 24 Game

9.1.1 Deepseek-CoderV2

Listing 16: Without partial successor soundness test

```
AutoToS prompt:

The 24 Game is a mathematical card game in which the objective is to
    find a way to manipulate four integers so that the end result is
    24. The game is played with a list of four numbers, and the player
     must use all four numbers exactly once, using any combination of
    addition, subtraction, multiplication, or division, to arrive at
    the number 24. If a state is defined by a list of numbers (4 or
    less), how are successor states defined? Please think step by step
    . Then, provide only the Python function that returns a list of
    successor states for an input state.

Here is an example of an initial state:
[6, 6, 2, 1]

Model response:
To define successor states for the 24 Game, we need to consider all
    possible ways to combine two numbers from the list using the
    allowed operations (addition, subtraction, multiplication, and
    division). Each combination will result in a new list of numbers,
    which represents a successor state.

Here's a step-by-step breakdown of how to generate successor states:

1. **Select Two Numbers**: Choose any two numbers from the list.
2. **Apply Operations**: Apply each of the four operations (addition,
    subtraction, multiplication, and division) to the selected numbers
    .
3. **Generate New List**: Replace the two selected numbers with the
    result of the operation, forming a new list of numbers.
4. **Repeat**: Repeat the process for all possible pairs of numbers
    and all operations.

For example, given the initial state `[6, 6, 2, 1]`:
- Select `6` and `6`, apply addition to get `12`, resulting in the new
     list `[12, 2, 1]`.
- Select `6` and `2`, apply subtraction to get `4`, resulting in the
    new list `[6, 4, 1]`.
- And so on for all pairs and operations.

Here is the Python function that returns a list of successor states
    for an input state:

python
```
def successor_states(state):
    successors = []
    n = len(state)
```
```

```
    # Iterate over all pairs of numbers in the state
    for i in range(n):
        for j in range(n):
            if i != j:
                a, b = state[i], state[j]

                # Generate successor states for each operation
                # Addition
                new_state = state[:]
                new_state.remove(a)
                new_state.remove(b)
                new_state.append(a + b)
                successors.append(new_state)

                # Subtraction (consider both a - b and b - a)
                new_state = state[:]
                new_state.remove(a)
                new_state.remove(b)
                new_state.append(a - b)
                successors.append(new_state)

                new_state = state[:]
                new_state.remove(a)
                new_state.remove(b)
                new_state.append(b - a)
                successors.append(new_state)

                # Multiplication
                new_state = state[:]
                new_state.remove(a)
                new_state.remove(b)
                new_state.append(a * b)
                successors.append(new_state)

                # Division (consider both a / b and b / a, if non-zero
                     denominator)
                if b != 0:
                    new_state = state[:]
                    new_state.remove(a)
                    new_state.remove(b)
                    new_state.append(a / b)
                    successors.append(new_state)

                if a != 0:
                    new_state = state[:]
                    new_state.remove(a)
                    new_state.remove(b)
                    new_state.append(b / a)
                    successors.append(new_state)

    return successors

# Example usage:
initial_state = [6, 6, 2, 1]
print(successor_states(initial_state))
```

This function generates all possible successor states by applying each operation to every pair of numbers in the input state. The resulting lists of numbers are collected in the 'successors' list and returned

AutoToS prompt:

```
Provide only the python code that tests whether a state is a goal
    state.
Example goal state: [24]
Example non-goal state: [24,1]

Model response:
python
```
def is_goal_state(state):
    return state == [24]

# Example usage:
goal_state = [24]
non_goal_state = [24, 1]

print(is_goal_state(goal_state))   # Output: True
print(is_goal_state(non_goal_state))   # Output: False
``

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 1352, failure: 0, total expansions 628081, total generated
    5420036
```

Listing 17: With partial successor soundness test

```
AutoToS prompt:

The 24 Game is a mathematical card game in which the objective is to
    find a way to manipulate four integers so that the end result is
    24. The game is played with a list of four numbers, and the player
     must use all four numbers exactly once, using any combination of
    addition, subtraction, multiplication, or division, to arrive at
    the number 24. If a state is defined by a list of numbers (4 or
    less), how are successor states defined? Please think step by step
    . Then, provide only the Python function that returns a list of
    successor states for an input state.

Here is an example of an initial state:
[6, 6, 2, 1]

Model response:
To define successor states for the 24 Game, we need to consider all
    possible ways to combine two numbers from the list using the
    allowed operations (addition, subtraction, multiplication, and
    division). Each combination will result in a new list of numbers,
    which represents a successor state.

Here's a step-by-step breakdown of how to generate successor states:

Select two numbers from the list: For a list of four numbers, there
    are 6 possible pairs of numbers to choose from.

Apply the operations: For each pair, apply addition, subtraction,
    multiplication, and division to get a new number.

Form the new list: Replace the two selected numbers with the result of
     the operation, resulting in a new list of three numbers.
```
```

Repeat for all pairs and operations: Generate all possible new lists by repeating the above steps for each pair and each operation.

Here is the Python function that returns a list of successor states for an input state:

python
Copy code
```
from itertools import combinations

def successor_states(state):
    successors = []
    for (a, b) in combinations(state, 2):
        remaining = [x for x in state if x not in (a, b)]
        successors.append(remaining + [a + b])
        successors.append(remaining + [a - b])
        successors.append(remaining + [b - a])
        successors.append(remaining + [a * b])
        if b != 0:
            successors.append(remaining + [a / b])
        if a != 0:
            successors.append(remaining + [b / a])
    return successors

# Example usage:
initial_state = [6, 6, 2, 1]
print(successor_states(initial_state))
```
Explanation of the Function:
combinations(state, 2): This generates all possible pairs of numbers from the input state.

remaining: This list contains the numbers that are not part of the current pair.

successors.append(remaining + [a + b]): This adds the new list formed by replacing the pair with the result of the operation to the list of successor states.

Division by zero check: To avoid division by zero errors, we check if the divisor is not zero before performing the division operation.

This function will generate all possible successor states by applying all allowed operations to each pair of numbers in the input state

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state.
Example goal state: [24]
Example non-goal state: [24,1]

Model response:
python
Copy code
```
def is_goal_state(state):
    return state == [24]

# Example usage:
goal_state = [24]
non_goal_state = [24, 1]

print(is_goal_state(goal_state))       # Output: True
```

```
print(is_goal_state(non_goal_state))   # Output: False
``

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
Invalid transformation: length mismatch - the length of a successor
    must be one less than the parent.
Let's think step by step. First think through in words why the
    successor function produced a successor that had a length that was
     not exactly one less than the parent. Then provide the complete
    Python code for the revised successor function that ensures the
    length of a successor is exactly one less than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.

Input state: [1, 1, 4, 6]
Example wrong successor state: [6, 5]

Model response:
Step-by-Step Analysis
The initial successor function produced successors that did not always
     have a length exactly one less than the parent due to the way it
    handled the operations and the formation of the new list.
    Specifically, the function did not correctly ensure that the new
    list always had one less element than the original list after
    combining two numbers.

To fix this, we need to:

Ensure that after combining two numbers with an operation, the
    resulting list has exactly one less element than the original list
    .

Properly handle the formation of the new list by replacing the two
    selected numbers with the result of the operation.

Revised Successor Function
Here is the revised Python function that ensures the length of a
    successor is exactly one less than the parent:

python
Copy code
```
from itertools import combinations

def successor_states(state):
    successors = []
    for (a, b) in combinations(state, 2):
        remaining = [x for x in state if x not in (a, b)]
        new_state = remaining + [a + b]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
        new_state = remaining + [a - b]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
        new_state = remaining + [b - a]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
        new_state = remaining + [a * b]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
```
```

```
        if b != 0:
            new_state = remaining + [a / b]
            if len(new_state) == len(state) - 1:
                successors.append(new_state)
        if a != 0:
            new_state = remaining + [b / a]
            if len(new_state) == len(state) - 1:
                successors.append(new_state)
    return successors

# Example usage:
initial_state = [1, 1, 4, 6]
print(successor_states(initial_state))
```

Explanation of the Revised Function:
combinations(state, 2): This generates all possible pairs of numbers from the input state.

remaining: This list contains the numbers that are not part of the current pair.

new_state = remaining + [a + b]: This forms the new list by replacing the pair with the result of the operation.

if len(new_state) == len(state) - 1: This checks if the new state has exactly one less element than the original state before adding it to the list of successors.

Division by zero check: To avoid division by zero errors, we check if the divisor is not zero before performing the division operation.

This function ensures that each successor state has exactly one less element than the parent state, adhering to the rules of the 24 Game

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 2

AutoToS prompt:
Successor function when run on the state [1, 1, 4, 6] failed to produce all successors.
Missing successors are: [[1, 4, 7], [-5, 1, 4], [1, 5, 6], [0.25, 1, 6], [-3, 1, 6], [0.16666666666666666, 1, 4], [1, 3, 6], [1, 4, 5]]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
Step-by-Step Analysis
The initial successor function failed to produce all possible successors because it did not correctly handle the formation of the new list after combining two numbers with an operation. Specifically, the function did not ensure that all possible combinations of operations on pairs of numbers were considered, and it did not correctly handle the formation of the new list by replacing the two selected numbers with the result of the operation.

To fix this, we need to:

Ensure that all possible pairs of numbers are considered.

Ensure that all operations (addition, subtraction, multiplication, division) are applied to each pair.

Properly handle the formation of the new list by replacing the two selected numbers with the result of the operation.

Revised Successor Function

Here is the revised Python function that ensures all possible successors are generated:

python

Copy code

```
from itertools import combinations

def successor_states(state):
    successors = []
    for (a, b) in combinations(state, 2):
        remaining = [x for x in state if x not in (a, b)]
        successors.append(remaining + [a + b])
        successors.append(remaining + [a - b])
        successors.append(remaining + [b - a])
        successors.append(remaining + [a * b])
        if b != 0:
            successors.append(remaining + [a / b])
        if a != 0:
            successors.append(remaining + [b / a])
    return successors

# Example usage:
initial_state = [1, 1, 4, 6]
print(successor_states(initial_state))
```

Explanation of the Revised Function:

combinations(state, 2): This generates all possible pairs of numbers from the input state.

remaining: This list contains the numbers that are not part of the current pair.

successors.append(remaining + [a + b]): This forms the new list by replacing the pair with the result of the addition operation and adds it to the list of successors.

successors.append(remaining + [a - b]): This forms the new list by replacing the pair with the result of the subtraction operation (a - b) and adds it to the list of successors.

successors.append(remaining + [b - a]): This forms the new list by replacing the pair with the result of the subtraction operation (b - a) and adds it to the list of successors.

successors.append(remaining + [a * b]): This forms the new list by replacing the pair with the result of the multiplication operation and adds it to the list of successors.

if b != 0: successors.append(remaining + [a / b]): This forms the new list by replacing the pair with the result of the division operation (a / b) and adds it to the list of successors, ensuring that division by zero is avoided.

if a != 0: successors.append(remaining + [b / a]): This forms the new list by replacing the pair with the result of the division

operation (b / a) and adds it to the list of successors, ensuring that division by zero is avoided.

This function ensures that all possible successors are generated by considering all pairs of numbers and applying all allowed operations, adhering to the rules of the 24 Game

Successor Iteration 3

AutoToS prompt:

Invalid transformation: length mismatch - the length of a successor must be one less than the parent.

Let's think step by step. First think through in words why the successor function produced a successor that had a length that was not exactly one less than the parent. Then provide the complete Python code for the revised successor function that ensures the length of a successor is exactly one less than the parent.

Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: [1, 1, 4, 6]

Example wrong successor state: [6, 5]

Model response:

Step-by-Step Analysis

The initial successor function produced successors that did not always have a length exactly one less than the parent due to the way it handled the operations and the formation of the new list. Specifically, the function did not correctly ensure that the new list always had one less element than the original list after combining two numbers.

To fix this, we need to:

Ensure that after combining two numbers with an operation, the resulting list has exactly one less element than the original list .

Properly handle the formation of the new list by replacing the two selected numbers with the result of the operation.

Revised Successor Function

Here is the revised Python function that ensures the length of a successor is exactly one less than the parent:

python

Copy code

```
from itertools import combinations

def successor_states(state):
    successors = []
    for (a, b) in combinations(state, 2):
        remaining = [x for x in state if x not in (a, b)]
        new_state = remaining + [a + b]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
        new_state = remaining + [a - b]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
        new_state = remaining + [b - a]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
        new_state = remaining + [a * b]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
```

```
        if b != 0:
            new_state = remaining + [a / b]
            if len(new_state) == len(state) - 1:
                successors.append(new_state)
        if a != 0:
            new_state = remaining + [b / a]
            if len(new_state) == len(state) - 1:
                successors.append(new_state)
    return successors

# Example usage:
initial_state = [1, 1, 4, 6]
print(successor_states(initial_state))
```

Explanation of the Revised Function:

combinations(state, 2): This generates all possible pairs of numbers from the input state.

remaining: This list contains the numbers that are not part of the current pair.

new_state = remaining + [a + b]: This forms the new list by replacing the pair with the result of the operation.

if len(new_state) == len(state) - 1: This checks if the new state has exactly one less element than the original state before adding it to the list of successors.

Division by zero check: To avoid division by zero errors, we check if the divisor is not zero before performing the division operation.

This function ensures that each successor state has exactly one less element than the parent state, adhering to the rules of the 24 Game

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 4

AutoToS prompt:
Successor function when run on the state [1, 1, 4, 6] failed to produce all successors.
Missing successors are: [[1, 4, 7], [-5, 1, 4], [1, 5, 6], [0.25, 1, 6], [-3, 1, 6], [0.16666666666666666, 1, 4], [1, 3, 6], [1, 4, 5]]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
Step-by-Step Analysis
The initial successor function failed to produce all possible successors because it did not correctly handle the formation of the new list after combining two numbers with an operation. Specifically, the function did not ensure that all possible combinations of operations on pairs of numbers were considered, and it did not correctly handle the formation of the new list by replacing the two selected numbers with the result of the operation.

To fix this, we need to:

Ensure that all possible pairs of numbers are considered.

Ensure that all operations (addition, subtraction, multiplication, division) are applied to each pair.

Properly handle the formation of the new list by replacing the two selected numbers with the result of the operation.

Revised Successor Function

Here is the revised Python function that ensures all possible successors are generated:

python

Copy code

```
from itertools import combinations

def successor_states(state):
    successors = []
    for (a, b) in combinations(state, 2):
        remaining = [x for x in state if x not in (a, b)]
        successors.append(remaining + [a + b])
        successors.append(remaining + [a - b])
        successors.append(remaining + [b - a])
        successors.append(remaining + [a * b])
        if b != 0:
            successors.append(remaining + [a / b])
        if a != 0:
            successors.append(remaining + [b / a])
    return successors

# Example usage:
initial_state = [1, 1, 4, 6]
print(successor_states(initial_state))
```

Explanation of the Revised Function:

combinations(state, 2): This generates all possible pairs of numbers from the input state.

remaining: This list contains the numbers that are not part of the current pair.

successors.append(remaining + [a + b]): This forms the new list by replacing the pair with the result of the addition operation and adds it to the list of successors.

successors.append(remaining + [a - b]): This forms the new list by replacing the pair with the result of the subtraction operation (a - b) and adds it to the list of successors.

successors.append(remaining + [b - a]): This forms the new list by replacing the pair with the result of the subtraction operation (b - a) and adds it to the list of successors.

successors.append(remaining + [a * b]): This forms the new list by replacing the pair with the result of the multiplication operation and adds it to the list of successors.

if b != 0: successors.append(remaining + [a / b]): This forms the new list by replacing the pair with the result of the division operation (a / b) and adds it to the list of successors, ensuring that division by zero is avoided.

if a != 0: successors.append(remaining + [b / a]): This forms the new list by replacing the pair with the result of the division

operation (b / a) and adds it to the list of successors, ensuring that division by zero is avoided.

This function ensures that all possible successors are generated by considering all pairs of numbers and applying all allowed operations, adhering to the rules of the 24 Game
Successor Iteration 5

AutoToS prompt:
Invalid transformation: length mismatch - the length of a successor must be one less than the parent.
Let’s think step by step. First think through in words why the successor function produced a successor that had a length that was not exactly one less than the parent. Then provide the complete Python code for the revised successor function that ensures the length of a successor is exactly one less than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: [1, 1, 4, 6]
Example wrong successor state: [6, 5]

Model response:
Step-by-Step Analysis
The initial successor function produced successors that did not always have a length exactly one less than the parent due to the way it handled the operations and the formation of the new list. Specifically, the function did not correctly ensure that the new list always had one less element than the original list after combining two numbers.

To fix this, we need to:

Ensure that after combining two numbers with an operation, the resulting list has exactly one less element than the original list .

Properly handle the formation of the new list by replacing the two selected numbers with the result of the operation.

Revised Successor Function
Here is the revised Python function that ensures the length of a successor is exactly one less than the parent:

python
Copy code
```
from itertools import combinations

def successor_states(state):
    successors = []
    for (a, b) in combinations(state, 2):
        remaining = [x for x in state if x not in (a, b)]
        new_state = remaining + [a + b]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
        new_state = remaining + [a - b]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
        new_state = remaining + [b - a]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
        new_state = remaining + [a * b]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
```

```
        if b != 0:
            new_state = remaining + [a / b]
            if len(new_state) == len(state) - 1:
                successors.append(new_state)
        if a != 0:
            new_state = remaining + [b / a]
            if len(new_state) == len(state) - 1:
                successors.append(new_state)
    return successors

# Example usage:
initial_state = [1, 1, 4, 6]
print(successor_states(initial_state))
```

Explanation of the Revised Function:

combinations(state, 2): This generates all possible pairs of numbers from the input state.

remaining: This list contains the numbers that are not part of the current pair.

new_state = remaining + [a + b]: This forms the new list by replacing the pair with the result of the operation.

if len(new_state) == len(state) - 1: This checks if the new state has exactly one less element than the original state before adding it to the list of successors.

Division by zero check: To avoid division by zero errors, we check if the divisor is not zero before performing the division operation.

This function ensures that each successor state has exactly one less element than the parent state, adhering to the rules of the 24 Game

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 6

AutoToS prompt:
Successor function when run on the state [1, 1, 4, 6] failed to produce all successors.
Missing successors are: [[1, 4, 7], [-5, 1, 4], [1, 5, 6], [0.25, 1, 6], [-3, 1, 6], [0.16666666666666666, 1, 4], [1, 3, 6], [1, 4, 5]]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
Step-by-Step Analysis
The initial successor function failed to produce all possible successors because it did not correctly handle the formation of the new list after combining two numbers with an operation. Specifically, the function did not ensure that all possible combinations of operations on pairs of numbers were considered, and it did not correctly handle the formation of the new list by replacing the two selected numbers with the result of the operation.

To fix this, we need to:

Ensure that all possible pairs of numbers are considered.

Ensure that all operations (addition, subtraction, multiplication, division) are applied to each pair.

Properly handle the formation of the new list by replacing the two selected numbers with the result of the operation.

Revised Successor Function

Here is the revised Python function that ensures all possible successors are generated:

python
Copy code

```
from itertools import combinations

def successor_states(state):
    successors = []
    for (a, b) in combinations(state, 2):
        remaining = [x for x in state if x not in (a, b)]
        successors.append(remaining + [a + b])
        successors.append(remaining + [a - b])
        successors.append(remaining + [b - a])
        successors.append(remaining + [a * b])
        if b != 0:
            successors.append(remaining + [a / b])
        if a != 0:
            successors.append(remaining + [b / a])
    return successors

# Example usage:
initial_state = [1, 1, 4, 6]
print(successor_states(initial_state))
```

Explanation of the Revised Function:

combinations(state, 2): This generates all possible pairs of numbers from the input state.

remaining: This list contains the numbers that are not part of the current pair.

successors.append(remaining + [a + b]): This forms the new list by replacing the pair with the result of the addition operation and adds it to the list of successors.

successors.append(remaining + [a - b]): This forms the new list by replacing the pair with the result of the subtraction operation (a - b) and adds it to the list of successors.

successors.append(remaining + [b - a]): This forms the new list by replacing the pair with the result of the subtraction operation (b - a) and adds it to the list of successors.

successors.append(remaining + [a * b]): This forms the new list by replacing the pair with the result of the multiplication operation and adds it to the list of successors.

if b != 0: successors.append(remaining + [a / b]): This forms the new list by replacing the pair with the result of the division operation (a / b) and adds it to the list of successors, ensuring that division by zero is avoided.

if a != 0: successors.append(remaining + [b / a]): This forms the new list by replacing the pair with the result of the division

operation (b / a) and adds it to the list of successors, ensuring that division by zero is avoided.

This function ensures that all possible successors are generated by considering all pairs of numbers and applying all allowed operations, adhering to the rules of the 24 Game

Successor Iteration 7

AutoToS prompt:

Invalid transformation: length mismatch - the length of a successor must be one less than the parent.

Let's think step by step. First think through in words why the successor function produced a successor that had a length that was not exactly one less than the parent. Then provide the complete Python code for the revised successor function that ensures the length of a successor is exactly one less than the parent.

Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: [1, 1, 4, 6]

Example wrong successor state: [6, 5]

Model response:

Step-by-Step Analysis

The initial successor function produced successors that did not always have a length exactly one less than the parent due to the way it handled the operations and the formation of the new list. Specifically, the function did not correctly ensure that the new list always had one less element than the original list after combining two numbers.

To fix this, we need to:

Ensure that after combining two numbers with an operation, the resulting list has exactly one less element than the original list.

Properly handle the formation of the new list by replacing the two selected numbers with the result of the operation.

Revised Successor Function

Here is the revised Python function that ensures the length of a successor is exactly one less than the parent:

python
Copy code

```
from itertools import combinations

def successor_states(state):
    successors = []
    for (a, b) in combinations(state, 2):
        remaining = [x for x in state if x not in (a, b)]
        new_state = remaining + [a + b]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
        new_state = remaining + [a - b]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
        new_state = remaining + [b - a]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
        new_state = remaining + [a * b]
        if len(new_state) == len(state) - 1:
            successors.append(new_state)
```

```
        if b != 0:
            new_state = remaining + [a / b]
            if len(new_state) == len(state) - 1:
                successors.append(new_state)
        if a != 0:
            new_state = remaining + [b / a]
            if len(new_state) == len(state) - 1:
                successors.append(new_state)
    return successors

# Example usage:
initial_state = [1, 1, 4, 6]
print(successor_states(initial_state))
```

Explanation of the Revised Function:
combinations(state, 2): This generates all possible pairs of numbers from the input state.

remaining: This list contains the numbers that are not part of the current pair.

new_state = remaining + [a + b]: This forms the new list by replacing the pair with the result of the operation.

if len(new_state) == len(state) - 1: This checks if the new state has exactly one less element than the original state before adding it to the list of successors.

Division by zero check: To avoid division by zero errors, we check if the divisor is not zero before performing the division operation.

This function ensures that each successor state has exactly one less element than the parent state, adhering to the rules of the 24 Game

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 8

AutoToS prompt:
Successor function when run on the state [1, 1, 4, 6] failed to produce all successors.
Missing successors are: [[1, 4, 7], [-5, 1, 4], [1, 5, 6], [0.25, 1, 6], [-3, 1, 6], [0.16666666666666666, 1, 4], [1, 3, 6], [1, 4, 5]]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
Step-by-Step Analysis
The initial successor function failed to produce all possible successors because it did not correctly handle the formation of the new list after combining two numbers with an operation. Specifically, the function did not ensure that all possible combinations of operations on pairs of numbers were considered, and it did not correctly handle the formation of the new list by replacing the two selected numbers with the result of the operation.

To fix this, we need to:

Ensure that all possible pairs of numbers are considered.

Ensure that all operations (addition, subtraction, multiplication, division) are applied to each pair.

Properly handle the formation of the new list by replacing the two selected numbers with the result of the operation.

Revised Successor Function
Here is the revised Python function that ensures all possible successors are generated:

python
Copy code
```
from itertools import combinations

def successor_states(state):
    successors = []
    for (a, b) in combinations(state, 2):
        remaining = [x for x in state if x not in (a, b)]
        successors.append(remaining + [a + b])
        successors.append(remaining + [a - b])
        successors.append(remaining + [b - a])
        successors.append(remaining + [a * b])
        if b != 0:
            successors.append(remaining + [a / b])
        if a != 0:
            successors.append(remaining + [b / a])
    return successors

# Example usage:
initial_state = [1, 1, 4, 6]
print(successor_states(initial_state))
```
Explanation of the Revised Function:
combinations(state, 2): This generates all possible pairs of numbers from the input state.

remaining: This list contains the numbers that are not part of the current pair.

successors.append(remaining + [a + b]): This forms the new list by replacing the pair with the result of the addition operation and adds it to the list of successors.

successors.append(remaining + [a - b]): This forms the new list by replacing the pair with the result of the subtraction operation (a - b) and adds it to the list of successors.

successors.append(remaining + [b - a]): This forms the new list by replacing the pair with the result of the subtraction operation (b - a) and adds it to the list of successors.

successors.append(remaining + [a * b]): This forms the new list by replacing the pair with the result of the multiplication operation and adds it to the list of successors.

if b != 0: successors.append(remaining + [a / b]): This forms the new list by replacing the pair with the result of the division operation (a / b) and adds it to the list of successors, ensuring that division by zero is avoided.

if a != 0: successors.append(remaining + [b / a]): This forms the new list by replacing the pair with the result of the division

```
    operation (b / a) and adds it to the list of successors, ensuring
    that division by zero is avoided.

This function ensures that all possible successors are generated by
    considering all pairs of numbers and applying all allowed
    operations, adhering to the rules of the 24 Game
Successor Iteration 9

AutoToS prompt:
Invalid transformation: length mismatch - the length of a successor
    must be one less than the parent.
Let's think step by step. First think through in words why the
    successor function produced a successor that had a length that was
     not exactly one less than the parent. Then provide the complete
    Python code for the revised successor function that ensures the
    length of a successor is exactly one less than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.

Input state: [1, 1, 4, 6]
Example wrong successor state: [6, 5]

Model response:


System message:
Maximum iterations reached for completeness test. Process stopped.
```

9.1.2 GPT4-Mini

Listing 18: Without partial successor soundness test

```
AutoToS prompt:

The 24 Game is a mathematical card game in which the objective is to
    find a way to manipulate four integers so that the end result is
    24. The game is played with a list of four numbers, and the player
     must use all four numbers exactly once, using any combination of
    addition, subtraction, multiplication, or division, to arrive at
    the number 24. If a state is defined by a list of numbers (4 or
    less), how are successor states defined? Please think step by step
    . Then, provide only the Python function that returns a list of
    successor states for an input state.

Here is an example of an initial state:
[6, 6, 2, 1]

Model response:
```python
def successor_states(state):
    from itertools import permutations
    from operator import add, sub, mul, truediv

    def calculate(a, b, operation):
        if operation == add:
            return a + b
        elif operation == sub:
            return a - b
        elif operation == mul:
            return a * b
        elif operation == truediv:
            if b != 0:
                return a / b
            return None
```
```

```
    def generate_expressions(numbers):
        if len(numbers) == 1:
            yield numbers[0]
        else:
            for i in range(len(numbers)):
                for j in range(len(numbers)):
                    if i != j:
                        remaining = [numbers[k] for k in range(len(
                            numbers)) if k != i and k != j]
                        for result in generate_expressions(remaining):
                            for operation in (add, sub, mul, truediv):
                                if operation == sub:
                                    yield calculate(numbers[i],
                                        numbers[j], operation)
                                elif operation == truediv:
                                    if numbers[j] != 0:
                                        yield calculate(numbers[i],
                                            numbers[j], operation)
                                else:
                                    yield calculate(numbers[i],
                                        numbers[j], operation)

    successors = set()
    for perm in permutations(state):
        for value in generate_expressions(list(perm)):
            if value is not None:
                new_state = [v for v in perm if v not in {perm[0],
                    perm[1]}]
                if len(new_state) < 4:
                    new_state.append(value)
                    successors.add(tuple(sorted(new_state)))

    return [list(s) for s in successors]
```

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state.
Example goal state: [24]
Example non-goal state: [24,1]

Model response:

```python
def is_goal_state(state):
    return len(state) == 1 and state[0] == 24
```

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 1

AutoToS prompt:
Successor function when run on the state [1, 1, 4, 6] failed to produce all successors.
Missing successors are: [[1, 1, 10], [0.6666666666666666, 1, 1], [1, 4, 7], [-2, 1, 1], [-5, 1, 4], [1, 4, 6], [1, 1, 2], [1, 5, 6], [0.25, 1, 6], [-3, 1, 6], [0, 4, 6], [0.16666666666666666, 1, 4], [1, 1, 24], [1, 3, 6], [2, 4, 6], [1, 4, 5], [1, 1, 1.5]]

```
First think step by step why the successor function failed to produce
    all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.

Model response:
The successor function likely failed to generate all successors of the
     state due to the limited handling of operational combinations and
     the omission of necessary states that can arise from expressions.
     The function needs to consider all possible permutations of the
    numbers for every operation regardless of the order of operations
    and evaluate all combinations to ensure that all results get added
     to the successor states.

Here's the updated function:

```python
def successor_states(state):
    from itertools import permutations
    from operator import add, sub, mul, truediv

    def calculate(a, b, operation):
        if operation == add:
            return a + b
        elif operation == sub:
            return a - b
        elif operation == mul:
            return a * b
        elif operation == truediv:
            if b != 0:
                return a / b
            return None

    successors = set()
    n = len(state)

    for perm in permutations(state):
        # Generate all combinations of two numbers and combine with
            remaining numbers
        for i in range(n):
            for j in range(n):
                if i != j:
                    a, b = perm[i], perm[j]
                    remaining = [perm[k] for k in range(n) if k != i
                        and k != j]

                    for operation in (add, sub, mul, truediv):
                        result = calculate(a, b, operation)
                        if result is not None:
                            new_state = remaining + [result]
                            successors.add(tuple(sorted(new_state)))

    return [list(s) for s in successors]
```

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 1352, failure: 0, total expansions 630318, total generated
    2470303
```

Listing 19: With partial successor soundness test

AutoToS prompt:

```
The 24 Game is a mathematical card game in which the objective is to
    find a way to manipulate four integers so that the end result is
    24. The game is played with a list of four numbers, and the player
     must use all four numbers exactly once, using any combination of
    addition, subtraction, multiplication, or division, to arrive at
    the number 24. If a state is defined by a list of numbers (4 or
    less), how are successor states defined? Please think step by step
    . Then, provide only the Python function that returns a list of
    successor states for an input state.

Here is an example of an initial state:
[6, 6, 2, 1]
```

Model response:

```python
def get_successor_states(state):
    from itertools import permutations
    from operator import add, sub, mul, truediv

    def valid_expression(a, b, op):
        if op == truediv and b == 0:
            return None
        return op(a, b)

    def generate_expressions(n1, n2):
        return [
            (valid_expression(n1, n2, add), '+'),
            (valid_expression(n1, n2, sub), '-'),
            (valid_expression(n1, n2, mul), '*'),
            (valid_expression(n1, n2, truediv), '/')
        ]

    successor_states = set()

    for nums in permutations(state):
        for i in range(3):
            for j in range(3):
                if i != j:
                    n1, n2, n3, n4 = nums[i], nums[j], nums[3 - i - j
                        ], nums[6 - i - j]
                    expressions = generate_expressions(n1, n2)

                    for result, op in expressions:
                        if result is not None:
                            new_numbers = [result] + [nums[k] for k in
                                 range(4) if k != i and k != j]
                            successor_states.add(tuple(sorted(
                               new_numbers)))

    return [list(state) for state in successor_states]
```

AutoToS prompt:

```
Provide only the python code that tests whether a state is a goal
    state.
Example goal state: [24]
Example non-goal state: [24,1]
```

Model response:

```python
def is_goal_state(state):
    return len(state) == 1 and state[0] == 24
```

~~~
```

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
The following exception was produced when testing the successor function:
  File "<string>", line 24, in get_successor_states

tuple index out of range. Please fix this exception in the successor function.
Model response:
```python
def get_successor_states(state):
    from itertools import permutations
    from operator import add, sub, mul, truediv

    def valid_expression(a, b, op):
        if op == truediv and b == 0:
            return None
        return op(a, b)

    def generate_expressions(n1, n2):
        return [
            (valid_expression(n1, n2, add), '+'),
            (valid_expression(n1, n2, sub), '-'),
            (valid_expression(n1, n2, mul), '*'),
            (valid_expression(n1, n2, truediv), '/')
        ]

    successor_states = set()

    for nums in permutations(state):
        for i in range(4):
            for j in range(4):
                if i != j:
                    n1, n2 = nums[i], nums[j]
                    remaining = [nums[k] for k in range(4) if k != i and k != j]
                    expressions = generate_expressions(n1, n2)

                    for result, op in expressions:
                        if result is not None:
                            new_numbers = [result] + remaining
                            successor_states.add(tuple(sorted(new_numbers)))

    return [list(state) for state in successor_states]
```
Successor Iteration 2

AutoToS prompt:
The following exception was produced when testing the successor function:
  File "<string>", line 25, in <listcomp>

tuple index out of range. Please fix this exception in the successor function.
Model response:
```python
def get_successor_states(state):
    from itertools import permutations
~~~

````
    from operator import add, sub, mul, truediv

    def valid_expression(a, b, op):
        if op == truediv and b == 0:
            return None
        return op(a, b)

    def generate_expressions(n1, n2):
        return [
            (valid_expression(n1, n2, add), '+'),
            (valid_expression(n1, n2, sub), '-'),
            (valid_expression(n1, n2, mul), '*'),
            (valid_expression(n1, n2, truediv), '/')
        ]

    successor_states = set()

    for nums in permutations(state):
        for i in range(4):
            for j in range(i + 1, 4):
                n1, n2 = nums[i], nums[j]
                remaining = [nums[k] for k in range(4) if k != i and k
                     != j]
                expressions = generate_expressions(n1, n2)

                for result, op in expressions:
                    if result is not None:
                        new_numbers = [result] + remaining
                        successor_states.add(tuple(sorted(new_numbers)
                            ))

    return [list(state) for state in successor_states]
```
Successor Iteration 3

AutoToS prompt:
The following exception was produced when testing the successor
    function:
  File "<string>", line 24, in <listcomp>

tuple index out of range. Please fix this exception in the successor
    function.
Model response:
```python
def get_successor_states(state):
    from itertools import permutations
    from operator import add, sub, mul, truediv

    def valid_expression(a, b, op):
        if op == truediv and b == 0:
            return None
        return op(a, b)

    def generate_expressions(n1, n2):
        return [
            (valid_expression(n1, n2, add), '+'),
            (valid_expression(n1, n2, sub), '-'),
            (valid_expression(n1, n2, mul), '*'),
            (valid_expression(n1, n2, truediv), '/')
        ]

    successor_states = set()

    for nums in permutations(state):
        for i in range(4):
````

```
            for j in range(i + 1, 4):
                n1, n2 = nums[i], nums[j]
                remaining = [nums[k] for k in range(4) if k != i and k
                     != j]
                expressions = generate_expressions(n1, n2)

                for result, op in expressions:
                    if result is not None:
                        new_numbers = [result] + remaining
                        if len(new_numbers) > 4:
                            continue
                        successor_states.add(tuple(sorted(new_numbers)
                            ))

    return [list(state) for state in successor_states]
```

Successor Iteration 4

AutoToS prompt:
The following exception was produced when testing the successor
    function:
  File "<string>", line 24, in <listcomp>

tuple index out of range. Please fix this exception in the successor
    function.
Model response:

```python
def get_successor_states(state):
    from itertools import permutations
    from operator import add, sub, mul, truediv

    def valid_expression(a, b, op):
        if op == truediv and b == 0:
            return None
        return op(a, b)

    def generate_expressions(n1, n2):
        return [
            (valid_expression(n1, n2, add), '+'),
            (valid_expression(n1, n2, sub), '-'),
            (valid_expression(n1, n2, mul), '*'),
            (valid_expression(n1, n2, truediv), '/')
        ]

    successor_states = set()

    if len(state) < 2:
        return []

    for nums in permutations(state):
        for i in range(4):
            for j in range(4):
                if i != j:
                    n1, n2 = nums[i], nums[j]
                    remaining = [nums[k] for k in range(4) if k != i
                        and k != j]
                    expressions = generate_expressions(n1, n2)

                    for result, op in expressions:
                        if result is not None:
                            new_numbers = [result] + remaining
                            if len(new_numbers) <= 4:
                                successor_states.add(tuple(sorted(
                                    new_numbers)))
```

````
    return [list(state) for state in successor_states]
```
Successor Iteration 5

AutoToS prompt:
The following exception was produced when testing the successor
    function:
  File "<string>", line 28, in <listcomp>

tuple index out of range. Please fix this exception in the successor
    function.
Model response:
```python
def get_successor_states(state):
    from itertools import permutations
    from operator import add, sub, mul, truediv

    def valid_expression(a, b, op):
        if op == truediv and b == 0:
            return None
        return op(a, b)

    def generate_expressions(n1, n2):
        return [
            (valid_expression(n1, n2, add), ’+’),
            (valid_expression(n1, n2, sub), ’-’),
            (valid_expression(n1, n2, mul), ’*’),
            (valid_expression(n1, n2, truediv), ’/’)
        ]

    successor_states = set()

    if len(state) < 2:
        return []

    for nums in permutations(state):
        for i in range(4):
            for j in range(i + 1, 4):
                n1, n2 = nums[i], nums[j]
                remaining = [nums[k] for k in range(4) if k != i and k
                     != j]
                expressions = generate_expressions(n1, n2)

                for result, op in expressions:
                    if result is not None:
                        new_numbers = [result] + remaining
                        if len(new_numbers) <= 4:
                            successor_states.add(tuple(sorted(
                                new_numbers)))

    return [list(state) for state in successor_states if len(state) >
        0]
```
Successor Iteration 6

AutoToS prompt:
The following exception was produced when testing the successor
    function:
  File "<string>", line 27, in <listcomp>

tuple index out of range. Please fix this exception in the successor
    function.
Model response:
```python
def get_successor_states(state):
````

```
    from itertools import permutations
    from operator import add, sub, mul, truediv

    def valid_expression(a, b, op):
        if op == truediv and b == 0:
            return None
        return op(a, b)

    def generate_expressions(n1, n2):
        return [
            (valid_expression(n1, n2, add), '+'),
            (valid_expression(n1, n2, sub), '-'),
            (valid_expression(n1, n2, mul), '*'),
            (valid_expression(n1, n2, truediv), '/')
        ]

    successor_states = set()

    # Check if the state has fewer than 2 numbers.
    if len(state) < 2:
        return []

    for nums in permutations(state):
        for i in range(4):
            for j in range(i + 1, 4):
                n1, n2 = nums[i], nums[j]
                remaining = [nums[k] for k in range(4) if k != i and k
                     != j]
                expressions = generate_expressions(n1, n2)

                for result, _ in expressions:
                    if result is not None:
                        new_numbers = [result] + remaining
                        successor_states.add(tuple(sorted(new_numbers)
                            ))

    return [list(state) for state in successor_states if len(state) >
        0]
```

Successor Iteration 7

AutoToS prompt:
The following exception was produced when testing the successor function:
  File "<string>", line 28, in <listcomp>

tuple index out of range. Please fix this exception in the successor function.
Model response:

```python
def get_successor_states(state):
    from itertools import permutations
    from operator import add, sub, mul, truediv

    def valid_expression(a, b, op):
        if op == truediv and b == 0:
            return None
        return op(a, b)

    def generate_expressions(n1, n2):
        return [
            (valid_expression(n1, n2, add), '+'),
            (valid_expression(n1, n2, sub), '-'),
            (valid_expression(n1, n2, mul), '*'),
            (valid_expression(n1, n2, truediv), '/')
```

```
        ]

    successor_states = set()

    # Ensure there are at least 4 numbers to manipulate.
    if len(state) < 2:
        return []

    for nums in permutations(state):
        for i in range(4):
            for j in range(4):
                if i != j:
                    n1, n2 = nums[i], nums[j]
                    remaining = [nums[k] for k in range(4) if k != i
                        and k != j]
                    if len(remaining) == 2:
                        expressions = generate_expressions(n1, n2)

                        for result, _ in expressions:
                            if result is not None:
                                new_numbers = [result] + remaining
                                successor_states.add(tuple(sorted(
                                    new_numbers)))

    return [list(state) for state in successor_states]
```

Successor Iteration 8

AutoToS prompt:
The following exception was produced when testing the successor function:
  File "<string>", line 29, in <listcomp>

tuple index out of range. Please fix this exception in the successor function.
Model response:

```python
def get_successor_states(state):
    from itertools import permutations
    from operator import add, sub, mul, truediv

    def valid_expression(a, b, op):
        if op == truediv and b == 0:
            return None
        return op(a, b)

    def generate_expressions(n1, n2):
        return [
            (valid_expression(n1, n2, add), '+'),
            (valid_expression(n1, n2, sub), '-'),
            (valid_expression(n1, n2, mul), '*'),
            (valid_expression(n1, n2, truediv), '/')
        ]

    successor_states = set()

    # Ensure there are exactly 4 numbers to manipulate.
    if len(state) != 4:
        return []

    for nums in permutations(state):
        for i in range(4):
            for j in range(i + 1, 4):
                n1, n2 = nums[i], nums[j]
```

```
                remaining = [nums[k] for k in range(4) if k != i and k
                     != j]
                expressions = generate_expressions(n1, n2)

                for result, _ in expressions:
                    if result is not None:
                        new_numbers = [result] + remaining
                        successor_states.add(tuple(sorted(new_numbers)
                            ))

    return [list(state) for state in successor_states]
```

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 9

AutoToS prompt:
Successor function when run on the state [1, 4, 6] failed to produce all successors.
Missing successors are: [[4, 6]]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The issue with the existing successor function is that it only considers pairs of numbers for operations and then adds the result with the remaining numbers. However, it does not explore all combinations of the remaining numbers after getting a result from a pair. To generate complete successor states, we need to ensure that after an operation is performed, the result is combined with all possible combinations of the remaining numbers, including further operations.

This means:
1. After calculating a result from two selected numbers, we should consider this result along with the others for further calculations, using the available operations again.
2. Instead of restricting to just pairing without recursion, we should generate all possible combinations through additional recursive steps.

Here's how to implement this:

```python
def get_successor_states(state):
    from itertools import permutations
    from operator import add, sub, mul, truediv

    def valid_expression(a, b, op):
        if op == truediv and b == 0:
            return None
        return op(a, b)

    def generate_expressions(n1, n2):
        return [
            (valid_expression(n1, n2, add), '+'),
            (valid_expression(n1, n2, sub), '-'),
            (valid_expression(n1, n2, mul), '*'),
            (valid_expression(n1, n2, truediv), '/')
        ]
```

```
    successor_states = set()

    def generate_successors(current_state):
        # Add current state if it has valid length
        if len(current_state) == 1:
            successor_states.add(tuple(current_state))

        for i in range(len(current_state)):
            for j in range(len(current_state)):
                if i != j:
                    n1, n2 = current_state[i], current_state[j]
                    remaining = [current_state[k] for k in range(len(
                        current_state)) if k != i and k != j]
                    expressions = generate_expressions(n1, n2)

                    for result, _ in expressions:
                        if result is not None:
                            new_numbers = [result] + remaining
                            generate_successors(new_numbers)

    generate_successors(state)
    return [list(state) for state in successor_states]
```

In this revised function:
- I introduced a recursive ‘generate_successors‘ function to keep generating successors by carrying forward the newly created numbers after each operation.
- This handles multiple generations of outcomes by applying the operations on the intermediate results recursively until we are left with a single number or valid states.

System message:
Maximum iterations reached for completeness test. Process stopped.

9.1.3 GPT4

Listing 20: Without partial successor soundness test

```
AutoToS prompt:

The 24 Game is a mathematical card game in which the objective is to
    find a way to manipulate four integers so that the end result is
    24. The game is played with a list of four numbers, and the player
     must use all four numbers exactly once, using any combination of
    addition, subtraction, multiplication, or division, to arrive at
    the number 24. If a state is defined by a list of numbers (4 or
    less), how are successor states defined? Please think step by step
    . Then, provide only the Python function that returns a list of
    successor states for an input state.

Here is an example of an initial state:
[6, 6, 2, 1]

Model response:
To generate successor states, we need to take every pair of numbers
    from the list, apply one of the four operations (addition,
    subtraction, multiplication, division), and generate a new state
    with the result of that operation and the remaining numbers. Here’
    s how to implement this:

1. Iterate over all pairs of numbers in the list.
2. For each pair, perform all valid operations.
```

```
3. Generate new states with the result of the operation and the
   remaining numbers.
4. Return the list of all possible new states.

Here's the function:

```python
from itertools import permutations

def generate_successor_states(state):
    def valid_operations(a, b):
        ops = [(a + b), (a - b), (b - a), (a * b)]
        if b != 0:
            ops.append(a / b)
        if a != 0:
            ops.append(b / a)
        return ops

    def generate_new_states(state):
        n = len(state)
        new_states = []

        for i in range(n):
            for j in range(i + 1, n):
                for result in valid_operations(state[i], state[j]):
                    new_state = [state[k] for k in range(n) if k != i
                        and k != j] + [result]
                    new_states.append(new_state)

        return new_states

    return generate_new_states(state)
```

AutoToS prompt:

Provide only the python code that tests whether a state is a goal
   state.
Example goal state: [24]
Example non-goal state: [24,1]

Model response:
```python
def is_goal_state(state):
    return len(state) == 1 and state[0] == 24
```

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 1352, failure: 0, total expansions 627424, total generated
   2706100
```

Listing 21: With partial successor soundness test

```
AutoToS prompt:

The 24 Game is a mathematical card game in which the objective is to
   find a way to manipulate four integers so that the end result is
   24. The game is played with a list of four numbers, and the player
```

must use all four numbers exactly once, using any combination of addition, subtraction, multiplication, or division, to arrive at the number 24. If a state is defined by a list of numbers (4 or less), how are successor states defined? Please think step by step . Then, provide only the Python function that returns a list of successor states for an input state.

Here is an example of an initial state:
[6, 6, 2, 1]

Model response:
To generate the successor states, we need to evaluate all possible operations between any two numbers and form new states by replacing the two numbers with the result of their operation. We'll iterate through all pairs of numbers in the list and apply addition, subtraction, multiplication, and both division directions (if the second number is non-zero to avoid division by zero). Here is the Python function that implements this:

```python
def generate_successor_states(state):
    from itertools import permutations
    from operator import add, sub, mul, truediv

    def generate_new_state(numbers, x, y, operation):
        result = operation(x, y)
        new_numbers = [num for num in numbers if num != x and num != y
            ]
        new_numbers.append(result)
        return new_numbers

    operations = [add, sub, mul, truediv]
    successors = []

    for x, y in permutations(state, 2):
        for operation in operations:
            if operation == truediv and y == 0:
                continue  # skip division by zero
            try:
                new_state = generate_new_state(state, x, y, operation)
                if new_state not in successors:
                    successors.append(new_state)
            except ZeroDivisionError:
                pass
    return successors
```

This function will return all possible successor states from the given state of numbers by applying all operations between each pair of numbers. Note that we check for division by zero before performing the operation to avoid errors.

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state.
Example goal state: [24]
Example non-goal state: [24,1]

Model response:
```python
def is_goal_state(state):
    return state == [24]
```

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
Invalid transformation: length mismatch - the length of a successor must be one less than the parent.
Let's think step by step. First think through in words why the successor function produced a successor that had a length that was not exactly one less than the parent. Then provide the complete Python code for the revised successor function that ensures the length of a successor is exactly one less than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: [1, 1, 4, 6]
Example wrong successor state: [6, 5]

Model response:
The issue with the previous implementation was that it didn't correctly account for removing exactly two numbers and adding just one result for each operation between a pair of numbers. The new state should always have one less number than the original state. Here is the correct approach to ensure the length of the successor state is one less than the parent:

1. Select a pair of numbers from the state.
2. Apply each operation to the pair and get the result.
3. Form a new state by removing the selected pair of numbers and adding the result.

Here is the corrected function:

```python
def generate_successor_states(state):
    from itertools import combinations
    from operator import add, sub, mul, truediv

    def generate_new_state(numbers, x, y, operation):
        result = operation(x, y)
        new_numbers = [num for num in numbers if num != x and num != y
            ]
        new_numbers.append(result)
        return new_numbers

    operations = [add, sub, mul, truediv]
    successors = []

    for x, y in combinations(state, 2):
        for operation in operations:
            if operation == truediv:
                if y == 0:
                    continue  # skip division by zero
                new_state = generate_new_state(state, x, y, operation)
                if len(new_state) == len(state) - 1 and new_state not
                    in successors:
                    successors.append(new_state)
            else:
                new_state = generate_new_state(state, x, y, operation)
                if len(new_state) == len(state) - 1 and new_state not
                    in successors:
                    successors.append(new_state)
    return successors
```

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 2

AutoToS prompt:
Successor function when run on the state [1, 1, 4, 6] failed to produce all successors.
Missing successors are: [[1, 4, 7], [-5, 1, 4], [1, 1, 2], [1, 5, 6], [0.25, 1, 6], [-3, 1, 6], [0.16666666666666666, 1, 4], [1, 3, 6], [1, 4, 5], [1, 1, 1.5]]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The main reason the previous successor function failed to produce all successor states is due to the logic used to remove the pair of numbers from the current state. Specifically, the logic used for removing numbers was not robust enough to handle cases where numbers might be equal or where other nuances like ordering affected the removal operation.

To fix this, we need to rewrite the removal logic to ensure that the state is correctly updated by removing exactly one instance of each selected pair and then adding the result. Here's the corrected approach:

1. Select a pair of numbers from the state.
2. Apply each operation to the pair and get the result.
3. Form a new state by removing exactly the selected pair of numbers (considering possible duplicates) and adding the result.

Here is the corrected function:

```python
def generate_successor_states(state):
    from itertools import permutations
    from operator import add, sub, mul, truediv

    def generate_new_state(numbers, x, y, operation):
        result = operation(x, y)
        temp_numbers = numbers[:]  # create a copy to manipulate
        temp_numbers.remove(x)     # remove one instance of x
        temp_numbers.remove(y)     # remove one instance of y
        new_numbers = temp_numbers + [result]  # add the result of
            operation
        return new_numbers

    operations = [add, sub, mul, truediv]
    successors = []

    for x, y in permutations(state, 2):
        if y != 0 or y == 0 and operations.index(truediv) != -1:
            for operation in operations:
                if operation == truediv and y == 0:
                    continue  # skip division by zero
                try:
                    new_state = generate_new_state(state, x, y,
                        operation)
                    if new_state not in successors:
                        successors.append(new_state)
```

```
                except ZeroDivisionError:
                    pass
    return successors
```

```
This function ensures that each operation on a pair of numbers results
     in a successor state that has exactly one less number than the
    original state, and it correctly handles cases with duplicate
    numbers in the state.

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 1352, failure: 0, total expansions 611864, total generated
    2477716
```

9.1.4 Llama3.1 70B

Listing 22: Without partial successor soundness test

```
AutoToS prompt:

The 24 Game is a mathematical card game in which the objective is to
    find a way to manipulate four integers so that the end result is
    24. The game is played with a list of four numbers, and the player
     must use all four numbers exactly once, using any combination of
    addition, subtraction, multiplication, or division, to arrive at
    the number 24. If a state is defined by a list of numbers (4 or
    less), how are successor states defined? Please think step by step
    . Then, provide only the Python function that returns a list of
    successor states for an input state.

Here is an example of an initial state:
[6, 6, 2, 1]

Model response:
To define successor states, we'll consider all possible operations (+,
     -, x, /) between each pair of numbers in the input state. Here's
    the step-by-step process:
Choose two numbers from the input state.
Apply each of the four operations to the chosen numbers.
Replace the chosen numbers in the input state with the result of the
    operation.
Repeat steps 1-3 for all possible pairs of numbers in the input state.
Here's the Python function to generate successor states:
Python
```

```
def get_successor_states(state):
    successor_states = []
    for i in range(len(state)):
        for j in range(i+1, len(state)):
            a, b = state[i], state[j]
            new_state = state[:i] + state[i+1:j] + state[j+1:]
            successor_states.append(new_state + [a + b])
            successor_states.append(new_state + [a - b])
            successor_states.append(new_state + [b - a])
            successor_states.append(new_state + [a * b])
            if b != 0:
                successor_states.append(new_state + [a / b])
            if a != 0:
                successor_states.append(new_state + [b / a])
    return successor_states
```

```
This function generates all possible successor states by applying the
    four operations to each pair of numbers in the input state. Note
    that division by zero is avoided by checking if the divisor is non
    -zero

AutoToS prompt:

Provide only the python code that tests whether a state is a goal
    state.
Example goal state: [24]
Example non-goal state: [24,1]

Model response:
Here's the Python function to test whether a state is a goal state:
Python
```
def is_goal_state(state):
    return len(state) == 1 and state[0] == 24
``

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 1352, failure: 0, total expansions 627424, total generated
    2706100
```

Listing 23: With partial successor soundness test

```
AutoToS prompt:

The 24 Game is a mathematical card game in which the objective is to
    find a way to manipulate four integers so that the end result is
    24. The game is played with a list of four numbers, and the player
     must use all four numbers exactly once, using any combination of
    addition, subtraction, multiplication, or division, to arrive at
    the number 24. If a state is defined by a list of numbers (4 or
    less), how are successor states defined? Please think step by step
    . Then, provide only the Python function that returns a list of
    successor states for an input state.

Here is an example of an initial state:
[6, 6, 2, 1]

Model response:
Successor State Definition
To define successor states, we need to consider all possible
    operations (+, -, x, /) between each pair of numbers in the input
    state. For each operation, we'll create a new state by applying
    the operation to the two numbers and replacing them with the
    result in the original list.
Step-by-Step Process
Choose two numbers from the input state (e.g., a and b).
Apply each operation (+, -, x, /) to a and b, resulting in a new value
     c.
Create a new state by replacing a and b with c in the original list.
Repeat steps 1-3 for all possible pairs of numbers in the input state.
Python Function
Python
```
def get_successor_states(state):
```

````
    successor_states = []
    ops = ['+', '-', '*', '/']

    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            a, b = state[i], state[j]
            for op in ops:
                if op == '+':
                    c = a + b
                elif op == '-':
                    c = a - b
                elif op == '*':
                    c = a * b
                elif op == '/':
                    if b != 0:
                        c = a / b
                    else:
                        continue
                new_state = state[:i] + [c] + state[j + 1:]
                successor_states.append(new_state)

    return successor_states
```
You can call this function with your initial state to get the list of
    successor states:
Python
initial_state = [6, 6, 2, 1]
successor_states = get_successor_states(initial_state)
print(successor_states

AutoToS prompt:

Provide only the python code that tests whether a state is a goal
    state.
Example goal state: [24]
Example non-goal state: [24,1]

Model response:
Python
```
def is_goal_state(state):
    return len(state) == 1 and state[0] == 24
``

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
Invalid transformation: length mismatch - the length of a successor
    must be one less than the parent.
Let's think step by step. First think through in words why the
    successor function produced a successor that had a length that was
     not exactly one less than the parent. Then provide the complete
    Python code for the revised successor function that ensures the
    length of a successor is exactly one less than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.

Input state: [1, 1, 4, 6]
Example wrong successor state: [5, 6]

Model response:
````

The successor function produced a successor with a length that was not exactly one less than the parent because it was replacing two numbers with one result, but not removing the other numbers from the list.
Here's the revised Python code for the successor function:
Python
```
def get_successor_states(state):
    successor_states = []
    ops = ['+', '-', '*', '/']

    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            a, b = state[i], state[j]
            for op in ops:
                if op == '+':
                    c = a + b
                elif op == '-':
                    c = a - b
                elif op == '*':
                    c = a * b
                elif op == '/':
                    if b != 0:
                        c = a / b
                    else:
                        continue
                # Create a new state with the result and the remaining
                     numbers
                new_state = [x for k, x in enumerate(state) if k not
                    in (i, j)] + [c]
                successor_states.append(new_state)

    return successor_states
```
In this revised version, we create a new state by including the result c and the remaining numbers from the original state, excluding the two numbers a and b that were used to produce c. This ensures that the length of each successor state is exactly one less than the parent state

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 2

AutoToS prompt:
Successor function when run on the state [1, 1, 4, 6] failed to produce all successors.
Missing successors are: [[1, 1, 2], [1, 3, 6], [1, 4, 5], [1, 1, 1.5]]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The successor function failed to produce all successors because it only considered operations between two numbers in the order they appear in the state. It didn't consider operations between numbers in different orders (e.g., b - a instead of a - b) and didn't consider operations between the same number (e.g., a / a or a * a)
.
Here's the revised Python code for the successor function:
Python
```
```

```
def get_successor_states(state):
    successor_states = []
    ops = ['+', '-', '*', '/']

    for i in range(len(state)):
        for j in range(len(state)):
            if i != j:
                a, b = state[i], state[j]
                for op in ops:
                    if op == '+':
                        c = a + b
                    elif op == '-':
                        c = a - b
                        if a != b:
                            c2 = b - a
                            new_state = [x for k, x in enumerate(state
                                ) if k not in (i, j)] + [c2]
                            successor_states.append(new_state)
                    elif op == '*':
                        c = a * b
                    elif op == '/':
                        if b != 0:
                            c = a / b
                        else:
                            continue
                        if a != b and a % b == 0:
                            c2 = b / a
                            new_state = [x for k, x in enumerate(state
                                ) if k not in (i, j)] + [c2]
                            successor_states.append(new_state)
                    new_state = [x for k, x in enumerate(state) if k
                        not in (i, j)] + [c]
                    successor_states.append(new_state)

    return successor_states
```
In this revised version, we consider operations between numbers in different orders and operations between the same number. We also avoid division by zero and ensure that we don't add duplicate states

Successor Iteration 3

AutoToS prompt:

The following exception was produced when testing the successor function:

  File "<string>", line 28, in get_successor_states

division by zero. Please fix this exception in the successor function.

Model response:

Here's the revised Python code for the successor function, which handles the division by zero exception:

Python
```
def get_successor_states(state):
    successor_states = []
    ops = ['+', '-', '*', '/']

    for i in range(len(state)):
        for j in range(len(state)):
            if i != j:
                a, b = state[i], state[j]
                for op in ops:
                    if op == '+':
                        c = a + b
                    elif op == '-':
```

```
                        c = a - b
                        if a != b:
                            c2 = b - a
                            new_state = [x for k, x in enumerate(state
                                ) if k not in (i, j)] + [c2]
                            successor_states.append(new_state)
                    elif op == '*':
                        c = a * b
                    elif op == '/':
                        if b != 0:
                            c = a / b
                            if a != b and a % b == 0:
                                c2 = b / a
                                new_state = [x for k, x in enumerate(
                                    state) if k not in (i, j)] + [c2]
                                successor_states.append(new_state)
                        elif a != 0:
                            c = b / a
                            new_state = [x for k, x in enumerate(state
                                ) if k not in (i, j)] + [c]
                            successor_states.append(new_state)
                    new_state = [x for k, x in enumerate(state) if k
                        not in (i, j)] + [c]
                    successor_states.append(new_state)

    return successor_states
```

In this revised version, we added a check to ensure that we don't divide by zero. If b is zero, we check if a is also zero. If a is not zero, we can safely divide b by a. This handles the division by zero exception

Successor Iteration 4

AutoToS prompt:

The following exception was produced when testing the successor function:

  File "<string>", line 26, in get_successor_states

division by zero. Please fix this exception in the successor function.

Model response:

Here's the revised Python code for the successor function, which handles the division by zero exception:

Python

```
def get_successor_states(state):
    successor_states = []
    ops = ['+', '-', '*', '/']

    for i in range(len(state)):
        for j in range(len(state)):
            if i != j:
                a, b = state[i], state[j]
                for op in ops:
                    if op == '+':
                        c = a + b
                    elif op == '-':
                        c = a - b
                        if a != b:
                            c2 = b - a
                            new_state = [x for k, x in enumerate(state
                                ) if k not in (i, j)] + [c2]
                            successor_states.append(new_state)
                    elif op == '*':
                        c = a * b
                    elif op == '/':
```

```
                        if b != 0:
                            c = a / b
                            if a != b and a % b == 0:
                                c2 = b / a
                                new_state = [x for k, x in enumerate(
                                    state) if k not in (i, j)] + [c2]
                                successor_states.append(new_state)
                        else:
                            continue  # Skip division by zero
                    new_state = [x for k, x in enumerate(state) if k
                        not in (i, j)] + [c]
                    successor_states.append(new_state)

    return successor_states
```

In this revised version, I added an else clause to the if b != 0 check . If b is zero, the function will continue to the next iteration of the loop, skipping the division operation and avoiding the division by zero exception

Successor Iteration 5

AutoToS prompt:

The following exception was produced when testing the successor function:

  File "<string>", line 26, in get_successor_states

division by zero. Please fix this exception in the successor function.

Model response:

Here’s the revised Python code for the successor function, which handles the division by zero exception:

Python

```
def get_successor_states(state):
    successor_states = []
    ops = [’+’, ’-’, ’*’, ’/’]

    for i in range(len(state)):
        for j in range(len(state)):
            if i != j:
                a, b = state[i], state[j]
                for op in ops:
                    if op == ’+’:
                        c = a + b
                    elif op == ’-’:
                        c = a - b
                        if a != b:
                            c2 = b - a
                            new_state = [x for k, x in enumerate(state
                                ) if k not in (i, j)] + [c2]
                            successor_states.append(new_state)
                    elif op == ’*’:
                        c = a * b
                    elif op == ’/’:
                        if b != 0 and a != 0:  # Check both a and b
                            for zero
                            c = a / b
                            if a != b and a % b == 0:
                                c2 = b / a
                                new_state = [x for k, x in enumerate(
                                    state) if k not in (i, j)] + [c2]
                                successor_states.append(new_state)
                    else:
                        continue  # Skip division by zero
                    new_state = [x for k, x in enumerate(state) if k
                        not in (i, j)] + [c]
```

```
                        successor_states.append(new_state)

    return successor_states
```
In this revised version, I added an additional check a != 0 to ensure
    that we don't divide by zero when a is also zero. If either a or b
     is zero, the function will continue to the next iteration of the
    loop, skipping the division operation and avoiding the division by
     zero exception

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 1352, failure: 0, total expansions 614323, total generated
    4596510
```

### 9.1.5 Llama3.1 405B

Listing 24: Without partial successor soundness test

```
AutoToS prompt:

The 24 Game is a mathematical card game in which the objective is to
    find a way to manipulate four integers so that the end result is
    24. The game is played with a list of four numbers, and the player
     must use all four numbers exactly once, using any combination of
    addition, subtraction, multiplication, or division, to arrive at
    the number 24. If a state is defined by a list of numbers (4 or
    less), how are successor states defined? Please think step by step
    . Then, provide only the Python function that returns a list of
    successor states for an input state.

Here is an example of an initial state:
[6, 6, 2, 1]

Model response:
Successor States in the 24 Game
Definition
Given a state in the 24 Game, a successor state is one that results
    from applying a mathematical operation (+, -, *, /) to two numbers
     in the state, reducing the list of numbers by one.
Step-by-Step Process
Choose Two Numbers: Select two distinct numbers from the current state
    .
Apply Operations: For each pair of chosen numbers, apply all four
    mathematical operations (+, -, *, /) to generate new numbers.
Create Successor States: For each new number generated, create a new
    state by replacing the two chosen numbers with the new number,
    ensuring the new state has one less number than the original state
    .
Handle Division by Zero: Ignore division operations that would result
    in division by zero.
Python Function to Generate Successor States
```Python
def get_successor_states(state):
    """
    Generate all possible successor states for a given state in the 24
         Game.

    Parameters:
    state (list): A list of numbers representing the current state.

    Returns:
    list: A list of lists, where each sublist is a successor state.
```

```python
    """
    successor_states = []

    # Iterate over all pairs of distinct numbers in the state
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            # Extract the two chosen numbers
            num1, num2 = state[i], state[j]

            # Generate new numbers by applying mathematical operations
            new_numbers = [
                num1 + num2,
                num1 - num2,
                num2 - num1,
                num1 * num2,
            ]
            # Handle division, avoiding division by zero
            if num2 != 0:
                new_numbers.append(num1 / num2)
            if num1 != 0:
                new_numbers.append(num2 / num1)

            # Create successor states for each new number
            for new_num in new_numbers:
                successor_state = state[:i] + state[i + 1:j] + state[j
                     + 1:] + [new_num]
                successor_states.append(successor_state)

    return successor_states
```

```
# Example usage
initial_state = [6, 6, 2, 1]
successors = get_successor_states(initial_state)
for state in successors:
    print(state)
```

This function generates all possible successor states by applying mathematical operations to all pairs of distinct numbers in the input state, handling division by zero appropriately. The result is a comprehensive list of next possible states in the 24 Game, useful for exploring the game tree or implementing a solution strategy.

Generated by Meta Llama 3.1-405B

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state.

Example goal state: [24]

Example non-goal state: [24,1]

Model response:

Python Function to Check if a State is a Goal State

```Python
def is_goal_state(state):
    """
    Check if a given state is a goal state in the 24 Game.

    A goal state is defined as a state containing only the number 24.

    Parameters:
    state (list): A list of numbers representing the current state.
```

```
    Returns:
    bool: True if the state is a goal state, False otherwise.
    """
    return state == [24]
```
Example Usage:
```Python
# Test a goal state
print(is_goal_state([24]))  # Output: True

# Test a non-goal state
print(is_goal_state([24, 1]))  # Output: False
```

```
Generated by Meta Llama 3.1-405B

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 1352, failure: 0, total expansions 627424, total generated
    2706100
```

Listing 25: With partial successor soundness test

```
AutoToS prompt:

The 24 Game is a mathematical card game in which the objective is to
    find a way to manipulate four integers so that the end result is
    24. The game is played with a list of four numbers, and the player
     must use all four numbers exactly once, using any combination of
    addition, subtraction, multiplication, or division, to arrive at
    the number 24. If a state is defined by a list of numbers (4 or
    less), how are successor states defined? Please think step by step
    . Then, provide only the Python function that returns a list of
    successor states for an input state.

Here is an example of an initial state:
[6, 6, 2, 1]

Model response:
Successor States Definition
Given a state with 4 or less numbers, successor states are defined as
    the states that can be reached by applying a mathematical
    operation (+, -, *, /) to two numbers in the current state. The
    two numbers are then replaced by the result of the operation.
Here are the steps to generate successor states:
Choose two numbers from the current state.
Apply each of the four mathematical operations (+, -, *, /) to the two
     numbers.
For each operation, create a new state by replacing the two numbers
    with the result of the operation.
Add the new states to the list of successor states.
Python Function: Successor States
Python
```
```
import itertools
import operator

def get_successor_states(state):
    """
    Returns a list of successor states for the given state.
```

```
    Args:
        state (list): A list of numbers representing the current state
            .

    Returns:
        list: A list of lists, where each sublist is a successor state
            .
    """
    successor_states = []
    ops = {
        '+': operator.add,
        '-': operator.sub,
        '*': operator.mul,
        '/': operator.truediv
    }

    # Choose two numbers from the current state
    for i, j in itertools.combinations(range(len(state)), 2):
        num1, num2 = state[i], state[j]

        # Apply each mathematical operation to the two numbers
        for op, func in ops.items():
            try:
                result = func(num1, num2)
                # Create a new state by replacing the two numbers with
                     the result
                new_state = state[:i] + [result] + state[i+1:j] +
                    state[j+1:]
                successor_states.append(new_state)
                # Also try the operation with num2 as the first
                    operand
                result = func(num2, num1)
                new_state = state[:i] + [result] + state[i+1:j] +
                    state[j+1:]
                successor_states.append(new_state)
            except ZeroDivisionError:
                # Skip division by zero
                continue

    return successor_states
```

Example usage:
Python

```
initial_state = [6, 6, 2, 1]
successor_states = get_successor_states(initial_state)
print(successor_states)
```

Generated by Meta Llama 3.1-405

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state.
Example goal state: [24]
Example non-goal state: [24,1]

Model response:
Python

```
def is_goal_state(state):
    """
    Returns True if the state is a goal state, False otherwise.

    A goal state is a state with a single number, which is 24.
```

```
    Args:
        state (list): A list of numbers representing the state.

    Returns:
        bool: Whether the state is a goal state.
    """
    return len(state) == 1 and state[0] == 24
```
```
Generated by Meta Llama 3.1-405

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 1352, failure: 0, total expansions 628046, total generated
    3603139
```

9.2 BlocksWorld

9.2.1 Deepseek-CoderV2

Listing 26: Without partial successor soundness test

```
AutoToS prompt:

I am playing with a set of blocks where I need to arrange the blocks
    into stacks. Here are the actions I can do
   Pick up a block from the table
   Put down a block on the table
   Unstack a block from on top of another block
   Stack a block on top of another block

   I have the following restrictions on my actions:
   I can only pick up or unstack one block at a time.
   I can only pick up or unstack a block if my hand is empty.
   I can only pick up a block if the block is on the table and the
       block is clear. A block is clear if the block has no other
       blocks on top of it and if the block is not picked up.
   I can only unstack a block from on top of another block if the
       block I am unstacking was really on top of the other block.
   I can only unstack a block from on top of another block if the
       block I am unstacking is clear.
   Once I pick up a block, I am holding the block and it is no longer
       clear and no longer on the table.
   Once I unstack from on top of another block, I am holding the block
        and it is no longer clear. Instead, the other block becomes
       clear.

   I can only put down a block that I am holding.
   I can only stack a block on top of another block if I am holding
       the block being stacked.
   I can only stack a block on top of another block if the block onto
       which I am stacking the block is clear.
   Once I put down a block, my hand becomes empty, the block becomes
       clear, and it is now on the table.
   Once I stack a block on top of another block, the block on top
       becomes clear and the block under it is no longer clear.
If a state is defined by a dictionary of following predicates "clear
    block", "on-table block", "arm-empty", "holding block", and "on
    block1 block2", how are successor states defined?
Here is an example of the initial state:
```

```
Init: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding
    ': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
Please provide only the Python code for the successor function that
    returns a list of successor states.

Model response:
Here is the Python code for the successor function that returns a list
     of successor states based on the given actions and restrictions:

python
```
```
def successor(state):
    successors = []
    clear_blocks = state['clear']
    on_table_blocks = state['on-table']
    arm_empty = state['arm-empty']
    holding_block = state['holding']
    on_blocks = state['on']

    # Action: Pick up a block from the table
    if arm_empty and holding_block is None:
        for block in clear_blocks:
            if block in on_table_blocks:
                new_state = state.copy()
                new_state['clear'] = [b for b in clear_blocks if b !=
                    block]
                new_state['on-table'] = [b for b in on_table_blocks if
                     b != block]
                new_state['arm-empty'] = False
                new_state['holding'] = block
                successors.append(new_state)

    # Action: Put down a block on the table
    if holding_block is not None:
        new_state = state.copy()
        new_state['clear'] = clear_blocks + [holding_block]
        new_state['on-table'] = on_table_blocks + [holding_block]
        new_state['arm-empty'] = True
        new_state['holding'] = None
        successors.append(new_state)

    # Action: Unstack a block from on top of another block
    if arm_empty and holding_block is None:
        for block1, block2 in on_blocks:
            if block1 in clear_blocks:
                new_state = state.copy()
                new_state['clear'] = [b for b in clear_blocks if b !=
                    block1] + [block2]
                new_state['on'] = [t for t in on_blocks if t != (
                    block1, block2)]
                new_state['arm-empty'] = False
                new_state['holding'] = block1
                successors.append(new_state)

    # Action: Stack a block on top of another block
    if holding_block is not None:
        for block in clear_blocks:
            new_state = state.copy()
            new_state['clear'] = [b for b in clear_blocks if b !=
                block] + [holding_block]
            new_state['on'] = on_blocks + [(holding_block, block)]
            new_state['arm-empty'] = True
            new_state['holding'] = None
            successors.append(new_state)
```

```
    return successors

# Example usage:
init_state = {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, '
    holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
successor_states = successor(init_state)
for s in successor_states:
    print(s)
```

This function takes the current state as input and returns a list of possible successor states based on the available actions and restrictions. Each action is implemented as a separate block of code within the function, ensuring that the state transitions adhere to the rules provided

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state. The goal specifies a partial state, the facts that must be true in any full goal state. Assume that the goal configuration is provided in the same dictionary format.
Here's an example goal specification:
Goal: {'clear': [], 'on-table': [], 'on': [('b', 'c'), ('d', 'b')]}

Model response:
python

```
def is_goal_state(state, goal):
    for key in goal:
        if key == 'on':
            for block_pair in goal[key]:
                if block_pair not in state[key]:
                    return False
        else:
            for block in goal[key]:
                if block not in state[key]:
                    return False
    return True

# Example usage:
goal_state = {'clear': [], 'on-table': [], 'on': [('b', 'c'), ('d', 'b
    ')]}
current_state = {'clear': ['a'], 'on-table': ['a'], 'arm-empty': True,
     'holding': None, 'on': [('b', 'c'), ('d', 'b')]}
print(is_goal_state(current_state, goal_state))  # Output: True or
    False depending on the current state
``
```

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 502, failure: 0, total expansions 50484, total generated 130127

Listing 27: With partial successor soundness test

AutoToS prompt:

```
I am playing with a set of blocks where I need to arrange the blocks
    into stacks. Here are the actions I can do
   Pick up a block from the table
   Put down a block on the table
   Unstack a block from on top of another block
   Stack a block on top of another block

   I have the following restrictions on my actions:
   I can only pick up or unstack one block at a time.
   I can only pick up or unstack a block if my hand is empty.
   I can only pick up a block if the block is on the table and the
       block is clear. A block is clear if the block has no other
       blocks on top of it and if the block is not picked up.
   I can only unstack a block from on top of another block if the
       block I am unstacking was really on top of the other block.
   I can only unstack a block from on top of another block if the
       block I am unstacking is clear.
   Once I pick up a block, I am holding the block and it is no longer
       clear and no longer on the table.
   Once I unstack from on top of another block, I am holding the block
        and it is no longer clear. Instead, the other block becomes
       clear.

   I can only put down a block that I am holding.
   I can only stack a block on top of another block if I am holding
       the block being stacked.
   I can only stack a block on top of another block if the block onto
       which I am stacking the block is clear.
   Once I put down a block, my hand becomes empty, the block becomes
       clear, and it is now on the table.
   Once I stack a block on top of another block, the block on top
       becomes clear and the block under it is no longer clear.
If a state is defined by a dictionary of following predicates "clear
    block", "on-table block", "arm-empty", "holding block", and "on
    block1 block2", how are successor states defined?
Here is an example of the initial state:
Init: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding
    ': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
Please provide only the Python code for the successor function that
    returns a list of successor states.

Model response:
Here is the Python code for the successor function that returns a list
     of successor states based on the given actions and restrictions:

python
'''
def successor(state):
    successors = []
    clear_blocks = state['clear']
    on_table_blocks = state['on-table']
    arm_empty = state['arm-empty']
    holding_block = state['holding']
    on_blocks = state['on']

    # Action: Pick up a block from the table
    if arm_empty:
        for block in clear_blocks:
            if block in on_table_blocks:
                new_state = state.copy()
                new_state['clear'] = [b for b in clear_blocks if b !=
                    block]
                new_state['on-table'] = [b for b in on_table_blocks if
                     b != block]
                new_state['arm-empty'] = False
```

```
                new_state['holding'] = block
                successors.append(new_state)

    # Action: Unstack a block from on top of another block
    if arm_empty:
        for block1, block2 in on_blocks:
            if block1 in clear_blocks:
                new_state = state.copy()
                new_state['clear'] = [b for b in clear_blocks if b !=
                    block1] + [block2]
                new_state['on'] = [pair for pair in on_blocks if pair
                    != (block1, block2)]
                new_state['arm-empty'] = False
                new_state['holding'] = block1
                successors.append(new_state)

    # Action: Put down a block on the table
    if not arm_empty and holding_block is not None:
        new_state = state.copy()
        new_state['clear'] = clear_blocks + [holding_block]
        new_state['on-table'] = on_table_blocks + [holding_block]
        new_state['arm-empty'] = True
        new_state['holding'] = None
        successors.append(new_state)

    # Action: Stack a block on top of another block
    if not arm_empty and holding_block is not None:
        for block in clear_blocks:
            new_state = state.copy()
            new_state['clear'] = [b for b in clear_blocks if b !=
                block] + [holding_block]
            new_state['on'] = on_blocks + [(holding_block, block)]
            new_state['arm-empty'] = True
            new_state['holding'] = None
            successors.append(new_state)

    return successors

# Example usage:
initial_state = {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True,
     'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
successor_states = successor(initial_state)
for s in successor_states:
    print(s)
```

This function takes the current state as input and returns a list of possible successor states based on the available actions and restrictions. Each action is implemented according to the rules provided, and the function ensures that only valid actions are performed

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state. The goal specifies a partial state, the facts that must be true in any full goal state. Assume that the goal configuration is provided in the same dictionary format.
Here's an example goal specification:
Goal: {'clear': [], 'on-table': [], 'on': [('b', 'c'), ('d', 'b')]}

Model response:
Here is the Python code that tests whether a state is a goal state based on the provided goal specification:

````
python
```
def is_goal_state(state, goal):
    for key in goal:
        if key == 'clear':
            if not all(block in state['clear'] for block in goal['
                clear']):
                return False
        elif key == 'on-table':
            if not all(block in state['on-table'] for block in goal['
                on-table']):
                return False
        elif key == 'on':
            for block1, block2 in goal['on']:
                if (block1, block2) not in state['on']:
                    return False
        elif key == 'arm-empty':
            if state['arm-empty'] != goal['arm-empty']:
                return False
        elif key == 'holding':
            if state['holding'] != goal['holding']:
                return False
    return True

# Example usage:
goal_state = {'clear': [], 'on-table': [], 'on': [('b', 'c'), ('d', 'b
    ')]}
current_state = {'clear': ['a', 'd'], 'on-table': ['a', 'c'], 'arm-
    empty': True, 'holding': None, 'on': [('b', 'c'), ('d', 'b')]}
print(is_goal_state(current_state, goal_state))  # Output: True or
    False
```

This function compares the current state with the goal state and
    returns `True` if the current state satisfies all the conditions
    specified in the goal state, otherwise it returns `False`. The
    comparison is done for each key in the goal dictionary, ensuring
    that the current state matches the goal state for the specified
    predicates

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 502, failure: 0, total expansions 50484, total generated
    130127
````

9.2.2 GPT4-Mini

Listing 28: Without partial successor soundness test

```
AutoToS prompt:

I am playing with a set of blocks where I need to arrange the blocks
    into stacks. Here are the actions I can do
   Pick up a block from the table
   Put down a block on the table
   Unstack a block from on top of another block
   Stack a block on top of another block
```

```
    I have the following restrictions on my actions:
    I can only pick up or unstack one block at a time.
    I can only pick up or unstack a block if my hand is empty.
    I can only pick up a block if the block is on the table and the
        block is clear. A block is clear if the block has no other
        blocks on top of it and if the block is not picked up.
    I can only unstack a block from on top of another block if the
        block I am unstacking was really on top of the other block.
    I can only unstack a block from on top of another block if the
        block I am unstacking is clear.
    Once I pick up a block, I am holding the block and it is no longer
        clear and no longer on the table.
    Once I unstack from on top of another block, I am holding the block
         and it is no longer clear. Instead, the other block becomes
        clear.

    I can only put down a block that I am holding.
    I can only stack a block on top of another block if I am holding
        the block being stacked.
    I can only stack a block on top of another block if the block onto
        which I am stacking the block is clear.
    Once I put down a block, my hand becomes empty, the block becomes
        clear, and it is now on the table.
    Once I stack a block on top of another block, the block on top
        becomes clear and the block under it is no longer clear.
If a state is defined by a dictionary of following predicates "clear
    block", "on-table block", "arm-empty", "holding block", and "on
    block1 block2", how are successor states defined?
Here is an example of the initial state:
Init: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding
    ': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
Please provide only the Python code for the successor function that
    returns a list of successor states.

Model response:
'''python
def successor(state):
    from copy import deepcopy

    successors = []

    def can_pick_up(block):
        return block in state['on-table'] and block in state['clear']
            and state['arm-empty']

    def can_unstack(block1, block2):
        return (block1, block2) in state['on'] and block1 in state['
            clear'] and state['arm-empty']

    def can_stack(block1, block2):
        return block1 in state['holding'] and block2 in state['clear']

    for block in state['on-table']:
        if can_pick_up(block):
            new_state = deepcopy(state)
            new_state['holding'] = block
            new_state['arm-empty'] = False
            new_state['clear'].remove(block)
            new_state['on-table'].remove(block)
            successors.append(new_state)

    for block1, block2 in state['on']:
        if can_unstack(block1, block2):
            new_state = deepcopy(state)
            new_state['holding'] = block1
```

```
            new_state['arm-empty'] = False
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            new_state['clear'].remove(block1)
            successors.append(new_state)

    if state['holding'] is not None:
        block = state['holding']
        if block in state['clear']:
            new_state = deepcopy(state)
            new_state['on-table'].append(block)
            new_state['arm-empty'] = True
            new_state['holding'] = None
            new_state['clear'].append(block)
            successors.append(new_state)

        for block2 in state['clear']:
            if can_stack(block, block2):
                new_state = deepcopy(state)
                new_state['holding'] = None
                new_state['arm-empty'] = True
                new_state['on'].append((block, block2))
                new_state['clear'].remove(block2)
                successors.append(new_state)

    return successors
```

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state. The goal specifies a partial state, the facts that must be true in any full goal state. Assume that the goal configuration is provided in the same dictionary format.
Here's an example goal specification:
Goal: {'clear': [], 'on-table': [], 'on': [('b', 'c'), ('d', 'b')]}

Model response:
```python
def is_goal(state, goal):
    return (set(state['clear']) == set(goal['clear']) and
            set(state['on-table']) == set(goal['on-table']) and
            set(state['on']) == set(goal['on']))
```
Goal Iteration 1 - Goal Soundness Test

AutoToS prompt:
The goal test function failed on the following input state {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}, incorrectly reporting it as a non-goal state.
First think step by step what it means for a state to be a goal state in this domain. Then think through in words why the goal test function incorrectly reported input state: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]} as a non-goal state. Now, revise the goal test function and ensure it returns true for the input state.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.
Goal: {'clear': [], 'on-table': [], 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}

Model response:
To determine if a state is a goal state, we need to check whether the required conditions from the goal specification are satisfied. In

this case, we are looking for a state that matches the specified goal, which includes the contents of the ‘clear‘, ‘on-table‘, and ‘on‘ lists.

The previous goal test function incorrectly reported the input state as a non-goal state because it was looking for exact matches in the ‘clear‘, ‘on-table‘, and ‘on‘ attributes based on the specified goal. The goal was defined as having no clear blocks and no blocks on the table, while the input state had a clear block ('b') and a block on the table ('d').

To fix the function, it should check that each condition that is specified in the goal is satisfied, but only for the blocks that are present in the input state. We can modify the logic to simply check if any of the goal's requirements are matched by the state without using strict equality checks of lists.

Here's the revised function:

```python
def is_goal(state, goal):
    return (all(item in state['clear'] for item in goal['clear']) and
            all(item in state['on-table'] for item in goal['on-table
                ']) and
            all(item in state['on'] for item in goal['on']))
```

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 1

AutoToS prompt:
Successor function when run on the state {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]} failed to produce all successors.
Missing successors are: [{'clear': ['a', 'b'], 'on-table': ['d', 'b'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd')]}, {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd'), ('b', 'a')]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The previous successor function failed to produce all successors from the given state because it did not account for the scenario where the block being held can be put down on the table. It also did not properly explore the possibility of stacking the block being held on the available clear blocks. Each possibility of putting down or stacking the held block needs to be explicitly captured to ensure that all valid successor states are generated.

To fix the function, we need to ensure that:
1. We can always generate a successor state for putting the held block down on the table, adding it to the ‘on-table‘ list and making it clear.
2. We also need to generate successor states for stacking the held block on any other clear block present in the state.

Here’s the revised successor function:

```python
def successor(state):
    from copy import deepcopy

    successors = []

    def can_pick_up(block):
        return block in state[’on-table’] and block in state[’clear’]
            and state[’arm-empty’]

    def can_unstack(block1, block2):
        return (block1, block2) in state[’on’] and block1 in state[’
            clear’] and state[’arm-empty’]

    def can_stack(block1, block2):
        return block1 in state[’holding’] and block2 in state[’clear’]

    for block in state[’on-table’]:
        if can_pick_up(block):
            new_state = deepcopy(state)
            new_state[’holding’] = block
            new_state[’arm-empty’] = False
            new_state[’clear’].remove(block)
            new_state[’on-table’].remove(block)
            successors.append(new_state)

    for block1, block2 in state[’on’]:
        if can_unstack(block1, block2):
            new_state = deepcopy(state)
            new_state[’holding’] = block1
            new_state[’arm-empty’] = False
            new_state[’clear’].append(block2)
            new_state[’on’].remove((block1, block2))
            new_state[’clear’].remove(block1)
            successors.append(new_state)

    if state[’holding’] is not None:
        block = state[’holding’]

        # Put down the held block on the table
        new_state = deepcopy(state)
        new_state[’on-table’].append(block)
        new_state[’arm-empty’] = True
        new_state[’holding’] = None
        new_state[’clear’].append(block)
        successors.append(new_state)

        # Stack the held block onto each clear block
        for block2 in state[’clear’]:
            new_state = deepcopy(state)
            new_state[’on’].append((block, block2))
            new_state[’clear’].remove(block2)
            new_state[’holding’] = None
            new_state[’arm-empty’] = True
            successors.append(new_state)

    return successors
```

System message:
Successor States Soundness Test Passed
Completeness test

Successor Iteration 2

AutoToS prompt:
Successor function when run on the state {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]} failed to produce all successors.
Missing successors are: [{'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd'), ('b', 'a')]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The previous implementation of the successor function failed to produce all successors in a specific scenario because it did not consider that the block being held ('b') can be stacked on a block that is under it ('a'). Specifically, since 'b' was held and 'a' was clear, the function should have generated a successor state reflecting 'b' being stacked on 'a', which was not accounted for.

To ensure that we capture all possible successor states:
1. We must check if the held block can be stacked on any block that is clear, including the situation where it can stack onto a block that is already in the `on` states.
2. We will generate the successor states for stacking the held block onto any clear block while the block that is stacked on 'b' becomes no longer clear.

Here's the revised function that addresses this:

```python
def successor(state):
    from copy import deepcopy

    successors = []

    def can_pick_up(block):
        return block in state['on-table'] and block in state['clear']
            and state['arm-empty']

    def can_unstack(block1, block2):
        return (block1, block2) in state['on'] and block1 in state['
            clear'] and state['arm-empty']

    for block in state['on-table']:
        if can_pick_up(block):
            new_state = deepcopy(state)
            new_state['holding'] = block
            new_state['arm-empty'] = False
            new_state['clear'].remove(block)
            new_state['on-table'].remove(block)
            successors.append(new_state)

    for block1, block2 in state['on']:
        if can_unstack(block1, block2):
            new_state = deepcopy(state)
            new_state['holding'] = block1
            new_state['arm-empty'] = False
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            new_state['clear'].remove(block1)
            successors.append(new_state)
```

```
    if state['holding'] is not None:
        block = state['holding']

        # Put down the held block on the table
        new_state = deepcopy(state)
        new_state['on-table'].append(block)
        new_state['arm-empty'] = True
        new_state['holding'] = None
        new_state['clear'].append(block)
        successors.append(new_state)

        # Stack the held block onto any clear block
        for block2 in state['clear']:
            new_state = deepcopy(state)
            new_state['on'].append((block, block2))
            new_state['clear'].remove(block2)
            new_state['holding'] = None
            new_state['arm-empty'] = True
            successors.append(new_state)

        # Check if we can also stack the held block on top of a block
            that is under it
        for block1, block2 in state['on']:
            if block1 == block and block2 in state['clear']:
                new_state = deepcopy(state)
                new_state['on'].append((block, block2))
                new_state['clear'].remove(block2)
                new_state['holding'] = None
                new_state['arm-empty'] = True
                new_state['clear'].remove(block1)
                successors.append(new_state)

    return successors
```

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 3

AutoToS prompt:
Successor function when run on the state {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]} failed to produce all successors.
Missing successors are: [{'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd'), ('b', 'a')]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The failure of the previous implementations to produce all successors of the provided state indicates that I did not properly account for the condition where the held block ('b') can be stacked on top of another block ('a') that is clear.

Specifically, the successors should include the scenario where you stack the held block ('b') on the block ('a') since 'a' is clear in the given state. The previous code lacked a proper check to handle this action while ensuring the clear status and updating the states pertinent to that action.

To address this, we should incorporate logic to add this action explicitly. Here’s how to adjust the successor function:

1. Continue to check that the held block can be stacked on any clear block.
2. Add logic to stack the held block specifically on the clear block if it satisfies the conditions.
3. Ensure to update the clear status accordingly and remove the block from the clear status when it is involved in a stacking action.

Here’s the corrected version of the successor function:

```python
def successor(state):
    from copy import deepcopy

    successors = []

    def can_pick_up(block):
        return block in state['on-table'] and block in state['clear']
            and state['arm-empty']

    def can_unstack(block1, block2):
        return (block1, block2) in state['on'] and block1 in state['
            clear'] and state['arm-empty']

    for block in state['on-table']:
        if can_pick_up(block):
            new_state = deepcopy(state)
            new_state['holding'] = block
            new_state['arm-empty'] = False
            new_state['clear'].remove(block)
            new_state['on-table'].remove(block)
            successors.append(new_state)

    for block1, block2 in state['on']:
        if can_unstack(block1, block2):
            new_state = deepcopy(state)
            new_state['holding'] = block1
            new_state['arm-empty'] = False
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            new_state['clear'].remove(block1)
            successors.append(new_state)

    if state['holding'] is not None:
        block = state['holding']

        # Put down the held block on the table
        new_state = deepcopy(state)
        new_state['on-table'].append(block)
        new_state['arm-empty'] = True
        new_state['holding'] = None
        new_state['clear'].append(block)
        successors.append(new_state)

        # Stack the held block onto any clear block
        for block2 in state['clear']:
            new_state = deepcopy(state)
            new_state['on'].append((block, block2))
            new_state['clear'].remove(block2)
            new_state['holding'] = None
            new_state['arm-empty'] = True
            successors.append(new_state)
```

```
    # Check if we can stack the held block on top of another block
    for block2 in state['clear']:
        if state['holding'] == 'b' and block2 == 'a':  # Assuming 'b'
            can stack on 'a'
            new_state = deepcopy(state)
            new_state['on'].append((block, block2))
            new_state['clear'].remove(block2)
            new_state['holding'] = None
            new_state['arm-empty'] = True
            new_state['clear'].remove(block)
            successors.append(new_state)

    return successors
```

This updated function adds the logic necessary to capture the potential successor states properly, ensuring that if 'b' is being held, and 'a' is clear, we generate the correct state where 'b' is stacked on 'a'.

Successor Iteration 4

AutoToS prompt:

The following exception was produced when testing the successor function:

  File "<string>", line 59, in successor

list.remove(x): x not in list. Please fix this exception in the successor function.

Model response:

The error occurs because of attempting to remove a block from the list of clear blocks without ensuring that the block is present in the list before performing the removal. Specifically, if we're trying to remove the block we're holding (e.g., 'b' in this case) from the clear list, it might not be there, as the block being held is indeed not clear.

To avoid this issue, we need to adjust the code in two areas:

1. When stacking the held block, we shouldn't try to remove it from the clear list, since it's not clear when it's being held.
2. We must ensure that we handle the case of adding/removing blocks to /from the 'clear' list more robustly.

Here's the corrected version of the successor function that addresses these issues:

```python
def successor(state):
    from copy import deepcopy

    successors = []

    def can_pick_up(block):
        return block in state['on-table'] and block in state['clear']
            and state['arm-empty']

    def can_unstack(block1, block2):
        return (block1, block2) in state['on'] and block1 in state['
            clear'] and state['arm-empty']

    for block in state['on-table']:
        if can_pick_up(block):
            new_state = deepcopy(state)
            new_state['holding'] = block
            new_state['arm-empty'] = False
```

```
            new_state['clear'].remove(block)
            new_state['on-table'].remove(block)
            successors.append(new_state)

    for block1, block2 in state['on']:
        if can_unstack(block1, block2):
            new_state = deepcopy(state)
            new_state['holding'] = block1
            new_state['arm-empty'] = False
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            new_state['clear'].remove(block1)
            successors.append(new_state)

    if state['holding'] is not None:
        block = state['holding']

        # Put down the held block on the table
        new_state = deepcopy(state)
        new_state['on-table'].append(block)
        new_state['arm-empty'] = True
        new_state['holding'] = None
        new_state['clear'].append(block)
        successors.append(new_state)

        # Stack the held block onto any clear block
        for block2 in state['clear']:
            new_state = deepcopy(state)
            new_state['on'].append((block, block2))
            new_state['clear'].remove(block2)
            new_state['holding'] = None
            new_state['arm-empty'] = True
            successors.append(new_state)

    # Check if we can stack the held block on top of a clear block
    for block in state['clear']:
        if state['holding'] == 'b' and block == 'a':  # Assuming 'b' can stack on 'a'
            new_state = deepcopy(state)
            new_state['on'].append((state['holding'], block))
            new_state['clear'].remove(block)
            new_state['holding'] = None
            new_state['arm-empty'] = True
            successors.append(new_state)

    return successors
```

Key Changes:
- Removed the attempt to remove the held block from the `clear` list.
- Verified the condition under which we were attempting to remove the 'clear' blocks to ensure it remains valid.
- Streamlined the condition checking for specific blocks being stacked to align with the actual game rules.

This should resolve the exception and ensure that all valid successors of the state are correctly generated.

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 5

AutoToS prompt:

Successor function when run on the state {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]} failed to produce all successors.
Missing successors are: [{'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd'), ('b', 'a')]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The failure of the function to produce the correct successors is due to not correctly handling the scenario in which the held block (`'b'`) is being stacked on the clear block (`'a'`). This involves ensuring that when the held block is stacked, the state of the clear blocks is properly updated.

The current logic does not account for the case where you stack the block that you are currently holding (`'b'`) onto a clear block (`'a'`). The function should check that `'a'` is clear, then create a successor state where `'b'` is stacked on `'a'`, removing `'b'` from the `clear` list because it will no longer be clear once stacked.

Here's the revised version of the successor function that ensures we properly check and create the successor where `'b'` is stacked on `'a'`:

```python
def successor(state):
    from copy import deepcopy

    successors = []

    def can_pick_up(block):
        return block in state['on-table'] and block in state['clear']
            and state['arm-empty']

    def can_unstack(block1, block2):
        return (block1, block2) in state['on'] and block1 in state['
            clear'] and state['arm-empty']

    # Generate successors for picking up blocks
    for block in state['on-table']:
        if can_pick_up(block):
            new_state = deepcopy(state)
            new_state['holding'] = block
            new_state['arm-empty'] = False
            new_state['clear'].remove(block)
            new_state['on-table'].remove(block)
            successors.append(new_state)

    # Generate successors for unstacking blocks
    for block1, block2 in state['on']:
        if can_unstack(block1, block2):
            new_state = deepcopy(state)
            new_state['holding'] = block1
            new_state['arm-empty'] = False
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            new_state['clear'].remove(block1)
            successors.append(new_state)
```

```
    # If holding a block, explore possibilities
    if state['holding'] is not None:
        block = state['holding']

        # Put down the held block on the table
        new_state = deepcopy(state)
        new_state['on-table'].append(block)
        new_state['arm-empty'] = True
        new_state['holding'] = None
        new_state['clear'].append(block)
        successors.append(new_state)

        # Stack the held block onto any clear block
        for block2 in state['clear']:
            new_state = deepcopy(state)
            new_state['on'].append((block, block2))  # Stack 'block'
                on 'block2'
            new_state['clear'].remove(block2)       # 'block2' is no
                longer clear
            new_state['holding'] = None
            new_state['arm-empty'] = True
            successors.append(new_state)

    # Additionally check if we can stack held block onto the
        specifically clear block 'a'
    if state['holding'] == 'b' and 'a' in state['clear']:
        new_state = deepcopy(state)
        new_state['on'].append(('b', 'a'))  # Stack 'b' on 'a'
        new_state['clear'].remove('a')      # 'a' is no longer clear
        new_state['holding'] = None
        new_state['arm-empty'] = True
        successors.append(new_state)

    return successors
```

Key Updates:
1. **Stacking Logic:** The code now includes logic to stack `held` block (`'b'`) directly on block `'a'` if it's clear.
2. **State Management:** When stacking `'b'` on `'a'`, it properly updates the clear list and maintains the integrity of the remaining part of the state.

This should now produce the expected successors, handling the specific case of stacking `'b'` on `'a'` correctly.

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 6

AutoToS prompt:
Successor function when run on the state {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]} failed to produce all successors.
Missing successors are: [{'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd'), ('b', 'a')]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:

The failure of the successor function to produce the expected successor state arises from a misunderstanding of the circumstance when the held block (''b'') can be stacked onto another block ('' a''). The function did not produce the necessary state when placing the held block onto a block that is also in the state because it was checked separately instead of being a general case handled in the stacking loop.

The key points are:
1. When the block is stacked, we must ensure that the 'clear' characteristic of the block we are stacking onto gets updated correctly. In this case, when stacking ''b'' on top of ''a'', ''a '' becomes no longer clear.
2. The logic did not correctly account for situations where the stack action could resolve to a state that was valid by definition of the rules.

To fix the function, we need to make sure the stack operation correctly identifies that while holding block ''b'', it can indeed be stacked on 'clear' block ''a''.

Here's the corrected successor function:

```python
def successor(state):
    from copy import deepcopy

    successors = []

    def can_pick_up(block):
        return block in state['on-table'] and block in state['clear']
            and state['arm-empty']

    def can_unstack(block1, block2):
        return (block1, block2) in state['on'] and block1 in state['
            clear'] and state['arm-empty']

    # Generate successors for picking up blocks
    for block in state['on-table']:
        if can_pick_up(block):
            new_state = deepcopy(state)
            new_state['holding'] = block
            new_state['arm-empty'] = False
            new_state['clear'].remove(block)
            new_state['on-table'].remove(block)
            successors.append(new_state)

    # Generate successors for unstacking blocks
    for block1, block2 in state['on']:
        if can_unstack(block1, block2):
            new_state = deepcopy(state)
            new_state['holding'] = block1
            new_state['arm-empty'] = False
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            new_state['clear'].remove(block1)
            successors.append(new_state)

    # If holding a block, explore possibilities
    if state['holding'] is not None:
        block = state['holding']

        # Put down the held block on the table
        new_state = deepcopy(state)
        new_state['on-table'].append(block)
```

```
        new_state['arm-empty'] = True
        new_state['holding'] = None
        new_state['clear'].append(block)
        successors.append(new_state)

        # Stack the held block onto any clear block
        for block2 in state['clear']:
            new_state = deepcopy(state)
            new_state['on'].append((block, block2))  # Stack 'block'
                on 'block2'
            new_state['clear'].remove(block2)       # 'block2' is no
                longer clear
            new_state['holding'] = None
            new_state['arm-empty'] = True
            successors.append(new_state)

        # Specifically account for stacking 'b' on 'a' since 'a' is
            clear
        if state['holding'] == 'b' and 'a' in state['clear']:
            new_state = deepcopy(state)
            new_state['on'].append(('b', 'a'))  # Stack 'b' on 'a'
            new_state['clear'].remove('a')      # 'a' is no longer
                clear
            new_state['holding'] = None
            new_state['arm-empty'] = True
            successors.append(new_state)

    return successors
```

Key Changes:

- The function logic for stacking now incorporates explicit checks for the specific case of stacking `held` block 'b' on block 'a'.
- When the held block can be stacked on a clear block, the function effectively manages to reflect those changes in the `clear` list, ensuring that any block stacked on is properly marked as no longer clear.

This should now ensure that the successor reflects the correct output, capturing the state where 'b' is stacked on 'a' correctly.

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 7

AutoToS prompt:
Successor function when run on the state {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]} failed to produce all successors.
Missing successors are: [{'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd'), ('b', 'a')]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The failure to generate the expected successor state arises from incorrectly identifying the conditions under which the action of stacking the held block (`'b'`) on the clear block (`'a'`) can occur. Specifically, the function is failing to account for the

addition of the successor where ''b'' is stacked on ''a''
correctly.

Why the Failure Occurred:
1. The logic for stacking was not properly set up to include the specific condition of stacking the held block on top of a designated clear block.
2. There should be no assumptions outside of explicitly evaluating each potential stacking action when the block is being held.
3. After checking if the block being held can stack on a clear block, the necessary structural changes to the successor state must be made.

Correction Approach:
- We need to explicitly ensure that when the block being held is ''b'' and there exists a clear block ''a'', we generate a state where ''b'' is stacked on ''a''.
- Adjust the conditions under which we modify the list of clear blocks and the on relationship.

Here's the corrected implementation of the successor function:

```python
def successor(state):
    from copy import deepcopy

    successors = []

    def can_pick_up(block):
        return block in state['on-table'] and block in state['clear']
            and state['arm-empty']

    def can_unstack(block1, block2):
        return (block1, block2) in state['on'] and block1 in state['
            clear'] and state['arm-empty']

    # Generate successors for picking up blocks
    for block in state['on-table']:
        if can_pick_up(block):
            new_state = deepcopy(state)
            new_state['holding'] = block
            new_state['arm-empty'] = False
            new_state['clear'].remove(block)
            new_state['on-table'].remove(block)
            successors.append(new_state)

    # Generate successors for unstacking blocks
    for block1, block2 in state['on']:
        if can_unstack(block1, block2):
            new_state = deepcopy(state)
            new_state['holding'] = block1
            new_state['arm-empty'] = False
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            new_state['clear'].remove(block1)
            successors.append(new_state)

    # If holding a block, explore possibilities
    if state['holding'] is not None:
        block = state['holding']

        # Put down the held block on the table
        new_state = deepcopy(state)
        new_state['on-table'].append(block)
        new_state['arm-empty'] = True
```

```
        new_state['holding'] = None
        new_state['clear'].append(block)
        successors.append(new_state)

        # Stack the held block onto any clear block
        for block2 in state['clear']:
            new_state = deepcopy(state)
            new_state['on'].append((block, block2))  # Stack 'block' on 'block2'
            new_state['clear'].remove(block2)       # 'block2' is no longer clear
            new_state['holding'] = None
            new_state['arm-empty'] = True
            successors.append(new_state)

    # Specifically account for stacking 'b' on 'a' since 'a' is clear
    if state['holding'] == 'b' and 'a' in state['clear']:
        new_state = deepcopy(state)
        new_state['on'].append(('b', 'a'))  # Stack 'b' on 'a'
        new_state['clear'].remove('a')      # 'a' is no longer clear
        new_state['holding'] = None
        new_state['arm-empty'] = True
        successors.append(new_state)

    return successors
```

Key Changes Made:
- Explicitly accounted for the condition of stacking `'b'` on `'a'` if 'a' is clear and `'b'` is the held block.
- Updated the successors to ensure that the state of blocks is accurately reflected after stacking actions are performed.

This implementation should now correctly handle all scenarios of the provided state, including producing the expected successors like the one where `'b'` is stacked on `'a'`.

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 8

AutoToS prompt:
Successor function when run on the state {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]} failed to produce all successors.
Missing successors are: [{'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd'), ('b', 'a')]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
To address the failures observed when the successor function was run on the provided state, let's examine the specific requirements of the action and how the current function is designed.

Analysis of the Failure

1. **Understanding the Problem**: The state being analyzed is:
```
```

```
{'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding':
    'b', 'on': [('a', 'c'), ('c', 'd')]}
```
In this state, the following conditions hold:
- Block 'b' is currently being held.
- Block 'a' is clear, making it suitable for stacking.
- The function should generate a successor state where block 'b' is stacked onto block 'a'.

2. **Current Implementation Limitation**: The function handles multiple actions, but it does not explicitly ensure that when 'b' is held, a valid state is generated for stacking 'b' on 'a' when 'a' is clear. While it correctly allows for putting 'b' down on the table, it fails to generate the scenario of stacking 'b' on 'a'.

3. **Missing Successor**: The essential missing action is stacking 'b' on 'a' and then transitioning 'a' to no longer be clear. This needs to be explicitly handled.

Fixing the Function

We need to ensure:
- That when the held block is available for stacking on a clear block, especially if the clear block is a defined block like 'a', we create the successor state correctly.

Here's how the revised successor function looks:

```python
def successor(state):
    from copy import deepcopy

    successors = []

    def can_pick_up(block):
        return block in state['on-table'] and block in state['clear']
            and state['arm-empty']

    def can_unstack(block1, block2):
        return (block1, block2) in state['on'] and block1 in state['
            clear'] and state['arm-empty']

    # Generate successors for picking up blocks
    for block in state['on-table']:
        if can_pick_up(block):
            new_state = deepcopy(state)
            new_state['holding'] = block
            new_state['arm-empty'] = False
            new_state['clear'].remove(block)
            new_state['on-table'].remove(block)
            successors.append(new_state)

    # Generate successors for unstacking blocks
    for block1, block2 in state['on']:
        if can_unstack(block1, block2):
            new_state = deepcopy(state)
            new_state['holding'] = block1
            new_state['arm-empty'] = False
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            new_state['clear'].remove(block1)
            successors.append(new_state)

    # If holding a block, explore possibilities
    if state['holding'] is not None:
```

```
        block = state['holding']

        # Put down the held block on the table
        new_state = deepcopy(state)
        new_state['on-table'].append(block)  # Putting the held block
            on the table
        new_state['arm-empty'] = True
        new_state['holding'] = None
        new_state['clear'].append(block)  # The block is now clear
        successors.append(new_state)

        # Specify stacking action
        if block == 'b' and 'a' in state['clear']:
            new_state = deepcopy(state)
            new_state['on'].append(('b', 'a'))  # Stack 'b' on 'a'
            new_state['clear'].remove('a')      # 'a' is no longer
                clear after stacking
            new_state['holding'] = None
            new_state['arm-empty'] = True
            successors.append(new_state)

    return successors
```

Key Revisions Made:
- **Stacking Logic**: The function now includes specific logic to stack `'b'` on `'a'` if `'a'` is clear when `'b'` is the held block.
- **Successor State Update**: It ensures that when stacking occurs, the block that is stacked upon is marked as no longer clear.

This revised function should now properly generate all the expected successors for the given state, addressing the gaps in the previous implementation and ensuring that all valid actions are captured.

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 9

AutoToS prompt:
Successor function when run on the state {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]} failed to produce all successors.
Missing successors are: [{'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd'), ('b', 'a')]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
To identify why the successor function failed to produce the expected successors, we need to closely examine the logic of the current implementation and understand the actions available in the specified state.

Analyzing the Current Situation

Given the state:
```python
```

```
{'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]}
```

We need to identify possible actions that can produce valid successor states.

1. **Held Block:** The held block is `'b'`.
2. **Clear Block:** The only clear block is `'a'`.
3. **Possible Actions:**
   - **Put Down:** Block `'b'` can be put down on the table.
   - **Stacking:** Block `'b'` can be stacked on block `'a'` since it is clear.

The missing successor state is one where `'b'` is stacked on top of `'a'`, which should look like:

```python
{'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd'), ('b', 'a')]}
```

The Issue with Current Implementation

The previously defined successor function logic does not adequately account for the condition of stacking the held block `'b'` onto the clear block `'a'` when it is held. Although it handles the action of putting the block down, it seems to miss checking the stacking condition directly within the loop that processes holding blocks.

Correcting the Function

To ensure that the successor states include the action of stacking `'b'` on `'a'`, we'll explicitly check if the held block can be stacked onto a clear block, especially `'a'`.

Here's the corrected version of the successor function:

```python
def successor(state):
    from copy import deepcopy

    successors = []

    def can_pick_up(block):
        return block in state['on-table'] and block in state['clear']
            and state['arm-empty']

    def can_unstack(block1, block2):
        return (block1, block2) in state['on'] and block1 in state['
            clear'] and state['arm-empty']

    # Generate successors for picking up blocks
    for block in state['on-table']:
        if can_pick_up(block):
            new_state = deepcopy(state)
            new_state['holding'] = block
            new_state['arm-empty'] = False
            new_state['clear'].remove(block)
            new_state['on-table'].remove(block)
            successors.append(new_state)

    # Generate successors for unstacking blocks
    for block1, block2 in state['on']:
        if can_unstack(block1, block2):
            new_state = deepcopy(state)
```

```
            new_state['holding'] = block1
            new_state['arm-empty'] = False
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            new_state['clear'].remove(block1)
            successors.append(new_state)

    # If holding a block, explore possibilities
    if state['holding'] is not None:
        block = state['holding']

        # Action: Put down the held block on the table
        new_state = deepcopy(state)
        new_state['on-table'].append(block)
        new_state['arm-empty'] = True
        new_state['holding'] = None
        new_state['clear'].append(block)  # The block is now clear
            after being put down
        successors.append(new_state)

        # Action: Stack the held block onto any clear block ('a' in
            this case)
        if block == 'b' and 'a' in state['clear']:
            new_state = deepcopy(state)
            new_state['on'].append(('b', 'a'))  # Stack 'b' on 'a'
            new_state['clear'].remove('a')      # 'a' is no longer
                clear after stacking
            new_state['holding'] = None
            new_state['arm-empty'] = True
            successors.append(new_state)

    return successors
```

Key Changes Made:
1. **Direct Stacking Logic**: The function now includes a condition that checks if the held block (`'b'`) can be stacked on the clear block (`'a'`).
2. **Successor Generation for Stacking**: If the condition is met, it generates the appropriate successor state where `'b'` is stacked on `'a'`.

This implementation should now produce all the expected successors, including correctly stacking `'b'` on `'a'`.

System message:
Successor States Soundness Test Passed
Completeness test

System message:
Maximum iterations reached for completeness test. Process stopped.

Listing 29: With partial successor soundness test

AutoToS prompt:

I am playing with a set of blocks where I need to arrange the blocks into stacks. Here are the actions I can do
   Pick up a block from the table
   Put down a block on the table
   Unstack a block from on top of another block
   Stack a block on top of another block

   I have the following restrictions on my actions:
   I can only pick up or unstack one block at a time.

```
    I can only pick up or unstack a block if my hand is empty.
    I can only pick up a block if the block is on the table and the
        block is clear. A block is clear if the block has no other
        blocks on top of it and if the block is not picked up.
    I can only unstack a block from on top of another block if the
        block I am unstacking was really on top of the other block.
    I can only unstack a block from on top of another block if the
        block I am unstacking is clear.
    Once I pick up a block, I am holding the block and it is no longer
        clear and no longer on the table.
    Once I unstack from on top of another block, I am holding the block
         and it is no longer clear. Instead, the other block becomes
        clear.

    I can only put down a block that I am holding.
    I can only stack a block on top of another block if I am holding
        the block being stacked.
    I can only stack a block on top of another block if the block onto
        which I am stacking the block is clear.
    Once I put down a block, my hand becomes empty, the block becomes
        clear, and it is now on the table.
    Once I stack a block on top of another block, the block on top
        becomes clear and the block under it is no longer clear.
If a state is defined by a dictionary of following predicates "clear
    block", "on-table block", "arm-empty", "holding block", and "on
    block1 block2", how are successor states defined?
Here is an example of the initial state:
Init: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding
    ': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
Please provide only the Python code for the successor function that
    returns a list of successor states.

Model response:
'''python
def successor(state):
    from collections import defaultdict

    def can_pick_up(block):
        return block in state['on-table'] and block in state['clear']
            and state['arm-empty']

    def can_unstack(block1, block2):
        return (block1, block2) in state['on'] and block1 in state['
            clear']

    def can_stack(block1, block2):
        return block1 in state['holding'] and block2 in state['clear']

    successors = []

    # 1. Pick up a clear block from the table
    for block in state['on-table']:
        if can_pick_up(block):
            new_state = state.copy()
            new_state['on-table'].remove(block)
            new_state['holding'] = block
            new_state['arm-empty'] = False
            new_state['clear'].remove(block)
            successors.append(new_state)

    # 2. Unstack a block from on top of another block
    for block1, block2 in state['on']:
        if can_unstack(block1, block2):
            new_state = state.copy()
            new_state['holding'] = block1
```

~~~~
            new_state[’arm-empty’] = False
            new_state[’clear’].remove(block1)
            new_state[’clear’].append(block2)
            new_state[’on’].remove((block1, block2))
            successors.append(new_state)

    # 3. Put down the block being held onto the table
    if state[’holding’]:
        new_state = state.copy()
        block = state[’holding’]
        new_state[’holding’] = None
        new_state[’arm-empty’] = True
        new_state[’on-table’].append(block)
        new_state[’clear’].append(block)
        successors.append(new_state)

    # 4. Stack the block being held on top of a clear block
    for block in state[’clear’]:
        if block != state[’holding’]:
            new_state = state.copy()
            block_to_stack = state[’holding’]
            new_state[’holding’] = None
            new_state[’arm-empty’] = True
            new_state[’on’].append((block_to_stack, block))
            new_state[’clear’].remove(block)
            new_state[’clear’].append(block_to_stack)
            successors.append(new_state)

    return successors
```

AutoToS prompt:

Provide only the python code that tests whether a state is a goal
    state. The goal specifies a partial state, the facts that must be
    true in any full goal state. Assume that the goal configuration is
     provided in the same dictionary format.
Here’s an example goal specification:
Goal: {’clear’: [], ’on-table’: [], ’on’: [(’b’, ’c’), (’d’, ’b’)]}

Model response:
```python
def is_goal(state, goal):
    def matches(fact_list, goal_list):
        return all(item in fact_list for item in goal_list)

    return (matches(state[’clear’], goal[’clear’]) and
            matches(state[’on-table’], goal[’on-table’]) and
            matches(state[’on’], goal[’on’]))
```

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
The following exception was produced when testing the successor
    function:
  File "autoToS/src/domain_base_class.py", line 145, in _str
    val = sorted(val)
          ^^^^^^^^^^^

’<’ not supported between instances of ’NoneType’ and ’str’. Please
    fix this exception in the successor function.
~~~~

Model response:
```python
def successor(state):
    from collections import defaultdict

    def can_pick_up(block):
        return block in state['on-table'] and block in state['clear']
            and state['arm-empty']

    def can_unstack(block1, block2):
        return (block1, block2) in state['on'] and block1 in state['
            clear']

    successors = []

    # 1. Pick up a clear block from the table
    for block in state['on-table']:
        if can_pick_up(block):
            new_state = state.copy()
            new_state['on-table'].remove(block)
            new_state['holding'] = block
            new_state['arm-empty'] = False
            new_state['clear'].remove(block)
            successors.append(new_state)

    # 2. Unstack a block from on top of another block
    for block1, block2 in state['on']:
        if can_unstack(block1, block2):
            new_state = state.copy()
            new_state['holding'] = block1
            new_state['arm-empty'] = False
            new_state['clear'].remove(block1)
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            successors.append(new_state)

    # 3. Put down the block being held onto the table
    if state['holding']:
        new_state = state.copy()
        block = state['holding']
        new_state['holding'] = None
        new_state['arm-empty'] = True
        new_state['on-table'].append(block)
        new_state['clear'].append(block)
        successors.append(new_state)

    # 4. Stack the block being held on top of a clear block
    for block in state['clear']:
        if block != state['holding'] and state['holding'] is not None:
            new_state = state.copy()
            block_to_stack = state['holding']
            new_state['holding'] = None
            new_state['arm-empty'] = True
            new_state['on'].append((block_to_stack, block))
            new_state['clear'].remove(block)
            new_state['clear'].append(block_to_stack)
            successors.append(new_state)

    return successors
```
Successor Iteration 2

AutoToS prompt:
Input state should not change as a result of applying the successor
    function.

```
State before successor function applied: {'clear': ['b'], 'on-table':
    ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b
    ', 'a'), ('c', 'd')]}
State after successor function applied: {'clear': ['a'], 'on-table':
    ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c
    ', 'd')]}
Let's think step by step. First, think of the possible reasons why the
     input state may change as a result of applying the successor
    function, such as the use of shallow copy.
Then, provide the complete Python code for the revised successor
    function that returns a list of successor states and ensure it
    does not change the input state.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.

Model response:
The issue of the input state changing as a result of applying the
    successor function is likely due to the use of a shallow copy.
    When we create a shallow copy of the dictionary, the lists within
    the dictionary (like `on-table`, `clear`, and `on`) are not copied
     deeply; instead, they are still references to the original lists.
     Thus, when we modify these lists in the new state, it also
    modifies them in the original state.

To avoid this, we need to create deep copies of any mutable elements (
    like lists and tuples) in the state. Here's the revised successor
    function that ensures the input state does not change:

'''python
def successor(state):
    from collections import defaultdict
    from copy import deepcopy

    def can_pick_up(block):
        return block in state['on-table'] and block in state['clear']
            and state['arm-empty']

    def can_unstack(block1, block2):
        return (block1, block2) in state['on'] and block1 in state['
            clear']

    successors = []

    # 1. Pick up a clear block from the table
    for block in state['on-table']:
        if can_pick_up(block):
            new_state = deepcopy(state)  # Deep copy to avoid side
                effects
            new_state['on-table'].remove(block)
            new_state['holding'] = block
            new_state['arm-empty'] = False
            new_state['clear'].remove(block)
            successors.append(new_state)

    # 2. Unstack a block from on top of another block
    for block1, block2 in state['on']:
        if can_unstack(block1, block2):
            new_state = deepcopy(state)  # Deep copy to avoid side
                effects
            new_state['holding'] = block1
            new_state['arm-empty'] = False
            new_state['clear'].remove(block1)
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            successors.append(new_state)
```

```
    # 3. Put down the block being held onto the table
    if state['holding']:
        new_state = deepcopy(state)  # Deep copy to avoid side effects
        block = state['holding']
        new_state['holding'] = None
        new_state['arm-empty'] = True
        new_state['on-table'].append(block)
        new_state['clear'].append(block)
        successors.append(new_state)

    # 4. Stack the block being held on top of a clear block
    for block in state['clear']:
        if block != state['holding'] and state['holding'] is not None:
            new_state = deepcopy(state)  # Deep copy to avoid side
                effects
            block_to_stack = state['holding']
            new_state['holding'] = None
            new_state['arm-empty'] = True
            new_state['on'].append((block_to_stack, block))
            new_state['clear'].remove(block)
            new_state['clear'].append(block_to_stack)
            successors.append(new_state)

    return successors
```

```
System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 418, failure: 84, total expansions 102111, total generated
    273070
```

9.2.3 GPT4

Listing 30: Without partial successor soundness test

```
AutoToS prompt:

I am playing with a set of blocks where I need to arrange the blocks
    into stacks. Here are the actions I can do
   Pick up a block from the table
   Put down a block on the table
   Unstack a block from on top of another block
   Stack a block on top of another block

   I have the following restrictions on my actions:
   I can only pick up or unstack one block at a time.
   I can only pick up or unstack a block if my hand is empty.
   I can only pick up a block if the block is on the table and the
      block is clear. A block is clear if the block has no other
      blocks on top of it and if the block is not picked up.
   I can only unstack a block from on top of another block if the
      block I am unstacking was really on top of the other block.
   I can only unstack a block from on top of another block if the
      block I am unstacking is clear.
   Once I pick up a block, I am holding the block and it is no longer
      clear and no longer on the table.
   Once I unstack from on top of another block, I am holding the block
       and it is no longer clear. Instead, the other block becomes
      clear.

   I can only put down a block that I am holding.
```

I can only stack a block on top of another block if I am holding the block being stacked.
I can only stack a block on top of another block if the block onto which I am stacking the block is clear.
Once I put down a block, my hand becomes empty, the block becomes clear, and it is now on the table.
Once I stack a block on top of another block, the block on top becomes clear and the block under it is no longer clear.
If a state is defined by a dictionary of following predicates "clear block", "on-table block", "arm-empty", "holding block", and "on block1 block2", how are successor states defined?
Here is an example of the initial state:
Init: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
Please provide only the Python code for the successor function that returns a list of successor states.

Model response:
```python
def successor_states(state):
    from copy import deepcopy

    def is_action_valid(action, block=None, block1=None, block2=None):
        if action == "pick_up":
            return state['arm-empty'] and block in state['on-table']
                and block in state['clear']
        elif action == "put_down":
            return state['holding'] is not None
        elif action == "unstack":
            return state['arm-empty'] and (block1, block2) in state['
                on'] and block1 in state['clear']
        elif action == "stack":
            return state['holding'] is not None and block1 in state['
                clear']
        return False

    def apply_action(state, action, block=None, block1=None, block2=
        None):
        new_state = deepcopy(state)
        if action == "pick_up":
            new_state['on-table'].remove(block)
            new_state['clear'].remove(block)
            new_state['arm-empty'] = False
            new_state['holding'] = block
        elif action == "put_down":
            new_state['on-table'].append(state['holding'])
            new_state['clear'].append(state['holding'])
            new_state['arm-empty'] = True
            new_state['holding'] = None
        elif action == "unstack":
            new_state['on'].remove((block1, block2))
            new_state['clear'].append(block2)
            new_state['clear'].remove(block1)
            new_state['arm-empty'] = False
            new_state['holding'] = block1
        elif action == "stack":
            new_state['on'].append((state['holding'], block1))
            new_state['clear'].remove(block1)
            new_state['clear'].append(state['holding'])
            new_state['arm-empty'] = True
            new_state['holding'] = None
        return new_state

    successors = []
```

```
    # Generate all possible actions and generate new states
    for block in state['on-table']:
        if is_action_valid("pick_up", block=block):
            new_state = apply_action(state, "pick_up", block=block)
            successors.append(new_state)

    if state['holding'] is not None:
        new_state = apply_action(state, "put_down")
        successors.append(new_state)

    for (block1, block2) in state['on']:
        if is_action_valid("unstack", block1=block1, block2=block2):
            new_state = apply_action(state, "unstack", block1=block1,
                block2=block2)
            successors.append(new_state)

    if state['holding'] is not None:
        for clear_block in state['clear']:
            if is_action_valid("stack", block1=clear_block):
                new_state = apply_action(state, "stack", block1=
                    clear_block)
                successors.append(new_state)

    return successors
```

AutoToS prompt:

    Provide only the python code that tests whether a state is a goal
        state. The goal specifies a partial state, the facts that must
         be true in any full goal state. Assume that the goal
        configuration is provided in the same dictionary format.
    Here's an example goal specification:
    Goal: {'clear': [], 'on-table': [], 'on': [('b', 'c'), ('d', 'b')
        ]}

Model response:
```python
def is_goal_state(state, goal):
    def match(condition, state_values, goal_values):
        if isinstance(goal_values, list):
            return all(elem in state_values for elem in goal_values)
        return state_values == goal_values

    for predicate in goal:
        if not match(predicate, state[predicate], goal[predicate]):
            return False
    return True
```

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 502, failure: 0, total expansions 50456, total generated
    130124

Listing 31: With partial successor soundness test

AutoToS prompt:

```
I am playing with a set of blocks where I need to arrange the blocks
    into stacks. Here are the actions I can do
   Pick up a block from the table
   Put down a block on the table
   Unstack a block from on top of another block
   Stack a block on top of another block

   I have the following restrictions on my actions:
   I can only pick up or unstack one block at a time.
   I can only pick up or unstack a block if my hand is empty.
   I can only pick up a block if the block is on the table and the
       block is clear. A block is clear if the block has no other
       blocks on top of it and if the block is not picked up.
   I can only unstack a block from on top of another block if the
       block I am unstacking was really on top of the other block.
   I can only unstack a block from on top of another block if the
       block I am unstacking is clear.
   Once I pick up a block, I am holding the block and it is no longer
       clear and no longer on the table.
   Once I unstack from on top of another block, I am holding the block
        and it is no longer clear. Instead, the other block becomes
       clear.

   I can only put down a block that I am holding.
   I can only stack a block on top of another block if I am holding
       the block being stacked.
   I can only stack a block on top of another block if the block onto
       which I am stacking the block is clear.
   Once I put down a block, my hand becomes empty, the block becomes
       clear, and it is now on the table.
   Once I stack a block on top of another block, the block on top
       becomes clear and the block under it is no longer clear.
If a state is defined by a dictionary of following predicates "clear
    block", "on-table block", "arm-empty", "holding block", and "on
    block1 block2", how are successor states defined?
Here is an example of the initial state:
Init: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding
    ': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
Please provide only the Python code for the successor function that
    returns a list of successor states.

Model response:
'''python
def get_successor_states(state):
    def pick_up(block):
        if block in state['clear'] and block in state['on-table'] and
            state['arm-empty']:
            new_state = {
                'clear': state['clear'][:],
                'on-table': [b for b in state['on-table'] if b !=
                    block],
                'arm-empty': False,
                'holding': block,
                'on': [pair for pair in state['on']]
            }
            new_state['clear'].remove(block)
            return new_state
        return None

    def put_down(block):
        if state['holding'] == block:
            new_state = {
                'clear': state['clear'][:] + [block],
                'on-table': state['on-table'][:] + [block],
```

```
                'arm-empty': True,
                'holding': None,
                'on': [pair for pair in state['on']]
            }
            return new_state
        return None

    def unstack(block1, block2):
        if (block1, block2) in state['on'] and block1 in state['clear
            '] and state['arm-empty']:
            new_state = {
                'clear': state['clear'][:] + [block2],
                'on-table': state['on-table'][:],
                'arm-empty': False,
                'holding': block1,
                'on': [pair for pair in state['on'] if pair != (block1
                    , block2)]
            }
            new_state['clear'].remove(block1)
            return new_state
        return None

    def stack(block1, block2):
        if state['holding'] == block1 and block2 in state['clear']:
            new_state = {
                'clear': [b for b in state['clear'] if b != block2] +
                    [block1],
                'on-table': state['on-table'][:],
                'arm-empty': True,
                'holding': None,
                'on': state['on'][:] + [(block1, block2)]
            }
            return new_state
        return None

    successors = []

    for clear_block in state['clear']:
        if state['arm-empty']:
            if clear_block in state['on-table']:
                new_state = pick_up(clear_block)
                if new_state:
                    successors.append(new_state)
            for block1, block2 in state['on']:
                if block1 == clear_block:
                    new_state = unstack(block1, block2)
                    if new_state:
                        successors.append(new_state)
        else:
            if state['holding']:
                if clear_block != state['holding']:
                    new_state = stack(state['holding'], clear_block)
                    if new_state:
                        successors.append(new_state)

    if not state['arm-empty'] and state['holding']:
        new_state = put_down(state['holding'])
        if new_state:
            successors.append(new_state)

    return successors
```

AutoToS prompt:

```
    Provide only the python code that tests whether a state is a goal
        state. The goal specifies a partial state, the facts that must
         be true in any full goal state. Assume that the goal
        configuration is provided in the same dictionary format.
    Here's an example goal specification:
    Goal: {'clear': [], 'on-table': [], 'on': [('b', 'c'), ('d', 'b')
        ]}

Model response:
```python
def is_goal_state(state, goal):
    def match_predicate(predicate, state_value, goal_value):
        return all(item in state_value for item in goal_value)

    clear_match = match_predicate('clear', state.get('clear', []),
        goal.get('clear', []))
    on_table_match = match_predicate('on-table', state.get('on-table',
         []), goal.get('on-table', []))
    on_match = match_predicate('on', state.get('on', []), goal.get('on
        ', []))

    return clear_match and on_table_match and on_match
```

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 502, failure: 0, total expansions 221012, total generated
    729268
```

### 9.2.4 Llama3.1 70B

Listing 32: Without partial successor soundness test

```
AutoToS prompt:

I am playing with a set of blocks where I need to arrange the blocks
    into stacks. Here are the actions I can do
   Pick up a block from the table
   Put down a block on the table
   Unstack a block from on top of another block
   Stack a block on top of another block

   I have the following restrictions on my actions:
   I can only pick up or unstack one block at a time.
   I can only pick up or unstack a block if my hand is empty.
   I can only pick up a block if the block is on the table and the
       block is clear. A block is clear if the block has no other
       blocks on top of it and if the block is not picked up.
   I can only unstack a block from on top of another block if the
       block I am unstacking was really on top of the other block.
   I can only unstack a block from on top of another block if the
       block I am unstacking is clear.
   Once I pick up a block, I am holding the block and it is no longer
       clear and no longer on the table.
   Once I unstack from on top of another block, I am holding the block
        and it is no longer clear. Instead, the other block becomes
       clear.
```

```
   I can only put down a block that I am holding.
   I can only stack a block on top of another block if I am holding
      the block being stacked.
   I can only stack a block on top of another block if the block onto
      which I am stacking the block is clear.
   Once I put down a block, my hand becomes empty, the block becomes
      clear, and it is now on the table.
   Once I stack a block on top of another block, the block on top
      becomes clear and the block under it is no longer clear.
If a state is defined by a dictionary of following predicates "clear
    block", "on-table block", "arm-empty", "holding block", and "on
    block1 block2", how are successor states defined?
Here is an example of the initial state:
Init: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding
    ': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
Please provide only the Python code for the successor function that
    returns a list of successor states.

Model response:
Here is the Python code for the successor function:
Python
'''
def successor(state):
    successors = []
    if state['arm-empty']:
        # Pick up a block from the table
        for block in state['clear']:
            if block in state['on-table']:
                new_state = state.copy()
                new_state['arm-empty'] = False
                new_state['holding'] = block
                new_state['clear'].remove(block)
                new_state['on-table'].remove(block)
                successors.append(new_state)

        # Unstack a block from on top of another block
        for block1, block2 in state['on']:
            if block1 in state['clear']:
                new_state = state.copy()
                new_state['arm-empty'] = False
                new_state['holding'] = block1
                new_state['clear'].remove(block1)
                new_state['clear'].append(block2)
                successors.append(new_state)

    else:
        # Put down a block on the table
        new_state = state.copy()
        new_state['arm-empty'] = True
        new_state['on-table'].append(state['holding'])
        new_state['clear'].append(state['holding'])
        new_state['holding'] = None
        successors.append(new_state)

        # Stack a block on top of another block
        for block in state['clear']:
            new_state = state.copy()
            new_state['arm-empty'] = True
            new_state['on'].append((state['holding'], block))
            new_state['clear'].remove(block)
            new_state['holding'] = None
            successors.append(new_state)

    return successors
'''
```

This function takes a state as input and returns a list of successor states. It checks the current state and applies the possible actions to generate new states. Note that this code assumes that the input state is valid and follows the restrictions you provided

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state. The goal specifies a partial state, the facts that must be true in any full goal state. Assume that the goal configuration is provided in the same dictionary format.
Here's an example goal specification:
Goal: {'clear': [], 'on-table': [], 'on': [('b', 'c'), ('d', 'b')]}

Model response:
Here is the Python code that tests whether a state is a goal state:
Python
```
def is_goal_state(state, goal):
    for key, value in goal.items():
        if key == 'clear' and state['clear'] != value:
            return False
        elif key == 'on-table' and state['on-table'] != value:
            return False
        elif key == 'on' and set(state['on']) != set(value):
            return False
    return True
```
This function takes a state and a goal as input and returns True if the state is a goal state and False otherwise. It checks each predicate in the goal and verifies that it is satisfied in the state. Note that for the on predicate, it uses a set comparison to ignore the order of the tuples
Goal Iteration 1 - Goal Soundness Test

AutoToS prompt:
The goal test function failed on the following input state {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}, incorrectly reporting it as a non-goal state.
First think step by step what it means for a state to be a goal state in this domain. Then think through in words why the goal test function incorrectly reported input state: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]} as a non-goal state. Now, revise the goal test function and ensure it returns true for the input state.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.
Goal: {'clear': [], 'on-table': [], 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}

Model response:
Here's the revised goal test function:
Python
```
def is_goal_state(state, goal):
    for key, value in goal.items():
        if key == 'clear' and set(state['clear']) != set(value):
            return False
        elif key == 'on-table' and set(state['on-table']) != set(value
            ):
            return False
        elif key == 'on' and set(state['on']) != set(value):
            return False
    return True
```

```
Explanation:
A state is a goal state if all the predicates in the goal are
    satisfied.
The previous function incorrectly reported the input state as a non-
    goal state because it was checking for exact list equality, not
    set equality. This meant that the order of the elements in the
    lists mattered, which is not the case in this domain.
By changing the comparison to set equality, we ignore the order of the
     elements and correctly identify the input state as a goal state.
Note that I also added set comparison for the 'clear' and 'on-table'
    predicates, to be consistent with the 'on' predicate. This ensures
     that the order of the elements in these lists does not affect the
     goal test
Goal Iteration 2 - Goal Soundness Test

AutoToS prompt:
The goal test function failed on the following input state {'clear':
    ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on
    ': [('a', 'c'), ('b', 'a'), ('c', 'd')]}, incorrectly reporting it
     as a non-goal state.
First think step by step what it means for a state to be a goal state
    in this domain. Then think through in words why the goal test
    function incorrectly reported input state: {'clear': ['b'], 'on-
    table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c
    '), ('b', 'a'), ('c', 'd')]} as a non-goal state. Now, revise the
    goal test function and ensure it returns true for the input state.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.
Goal: {'clear': [], 'on-table': [], 'on': [('a', 'c'), ('b', 'a'), ('c
    ', 'd')]}

Model response:
To be a goal state, the following conditions must be met:
The 'clear' predicate must be empty, meaning no blocks are clear.
The 'on-table' predicate must be empty, meaning no blocks are on the
    table.
The 'on' predicate must match the goal configuration, meaning the
    blocks are stacked in the correct order.
The goal test function incorrectly reported the input state as a non-
    goal state because it was checking for exact equality with the
    goal state, including the 'clear' and 'on-table' predicates.
    However, the goal state only specifies that these predicates
    should be empty, not that they must be exactly equal to the input
    state.
Here is the revised goal test function:
Python
```
```
def is_goal_state(state, goal):
    for key, value in goal.items():
        if key == 'clear' and state['clear'] != []:
            return False
        elif key == 'on-table' and state['on-table'] != []:
            return False
        elif key == 'on' and set(state['on']) != set(value):
            return False
    return True
```
```
In this revised function, we only check that the 'clear' and 'on-table
    ' predicates are empty, and that the 'on' predicate matches the
    goal configuration. This ensures that the function returns True
    for the input state
Goal Iteration 3 - Goal Soundness Test

AutoToS prompt:
```

```
The goal test function failed on the following input state {'clear':
    ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on
    ': [('a', 'c'), ('b', 'a'), ('c', 'd')]}, incorrectly reporting it
     as a non-goal state.
First think step by step what it means for a state to be a goal state
    in this domain. Then think through in words why the goal test
    function incorrectly reported input state: {'clear': ['b'], 'on-
    table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c
    '), ('b', 'a'), ('c', 'd')]} as a non-goal state. Now, revise the
    goal test function and ensure it returns true for the input state.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.
Goal: {'clear': [], 'on-table': [], 'on': [('a', 'c'), ('b', 'a'), ('c
    ', 'd')]}

Model response:
To be a goal state, the following conditions must be met:
The 'clear' predicate must be empty, meaning no blocks are clear.
The 'on-table' predicate must be empty, meaning no blocks are on the
    table.
The 'on' predicate must match the goal configuration, meaning the
    blocks are stacked in the correct order.
The goal test function incorrectly reported the input state as a non-
    goal state because it was checking for exact equality with the
    goal state, including the 'clear' and 'on-table' predicates.
    However, the goal state only specifies that these predicates
    should be empty, not that they must be exactly equal to the input
    state.
Here is the revised goal test function:
Python
def is_goal_state(state, goal):
    for key, value in goal.items():
        if key == 'clear' and len(state['clear']) != len(value):
            return False
        elif key == 'on-table' and len(state['on-table']) != len(value
            ):
            return False
        elif key == 'on' and set(state['on']) != set(value):
            return False
    return True
In this revised function, we only check that the lengths of the 'clear
    ' and 'on-table' predicates match the goal state, and that the 'on
    ' predicate matches the goal configuration. This ensures that the
    function returns True for the input state.
However, this revised function still fails to return True for the
    input state because the 'clear' and 'on-table' predicates are not
    empty in the input state. To fix this, we need to revise the
    function to ignore the 'clear' and 'on-table' predicates if they
    are not present in the goal state.
Here is the revised goal test function:
Python
'''
def is_goal_state(state, goal):
    for key, value in goal.items():
        if key in state:
            if key == 'on' and set(state['on']) != set(value):
                return False
            elif key in ['clear', 'on-table'] and len(state[key]) !=
                len(value):
                return False
    return True
'''
In this revised function, we only check the predicates that are
    present in the goal state, and we ignore the 'clear' and 'on-table
```

' predicates if they are not present. This ensures that the function returns True for the input state

Goal Iteration 4 - Goal Soundness Test

AutoToS prompt:

The goal test function failed on the following input state {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}, incorrectly reporting it as a non-goal state.

First think step by step what it means for a state to be a goal state in this domain. Then think through in words why the goal test function incorrectly reported input state: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]} as a non-goal state. Now, revise the goal test function and ensure it returns true for the input state.

Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Goal: {'clear': [], 'on-table': [], 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}

Model response:

To be a goal state, the following conditions must be met:

The blocks are stacked in the correct order, as specified in the 'on' predicate.

The 'clear' and 'on-table' predicates are empty, meaning no blocks are clear or on the table.

The goal test function incorrectly reported the input state as a non-goal state because it was checking for exact equality with the goal state, including the 'clear' and 'on-table' predicates. However, the goal state only specifies that these predicates should be empty, but the input state has blocks 'b' and 'd' that are clear and on the table, respectively.

Here is the revised goal test function:

Python

```
def is_goal_state(state, goal):
    if 'on' in goal and set(state['on']) != set(goal['on']):
        return False
    if 'clear' in goal and len(state['clear']) != len(goal['clear']):
        return False
    if 'on-table' in goal and len(state['on-table']) != len(goal['on-table']):
        return False
    return True
```

In this revised function, we only check the predicates that are present in the goal state. If the 'on' predicate is present, we check that the blocks are stacked in the correct order. If the 'clear' or 'on-table' predicates are present, we check that they are empty. This ensures that the function returns True for the input state

Goal Iteration 5 - Goal Soundness Test

AutoToS prompt:

The goal test function failed on the following input state {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}, incorrectly reporting it as a non-goal state.

First think step by step what it means for a state to be a goal state in this domain. Then think through in words why the goal test function incorrectly reported input state: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]} as a non-goal state. Now, revise the goal test function and ensure it returns true for the input state.

Remember how you fixed the previous mistakes, if any. Keep the same function signature.
Goal: {'clear': [], 'on-table': [], 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}

Model response:
To be a goal state, the following conditions must be met:
The blocks are stacked in the correct order, as specified in the 'on' predicate.
The 'clear' and 'on-table' predicates are empty, meaning no blocks are clear or on the table.
The goal test function incorrectly reported the input state as a non-goal state because it was checking for exact equality with the goal state, including the 'clear' and 'on-table' predicates. However, the goal state only specifies that these predicates should be empty, but the input state has blocks 'b' and 'd' that are clear and on the table, respectively.
Here is the revised goal test function:
Python
```
def is_goal_state(state, goal):
    if 'on' in goal and set(state['on']) != set(goal['on']):
        return False
    if 'clear' in goal and goal['clear'] == [] and state['clear'] !=
        []:
        return False
    if 'on-table' in goal and goal['on-table'] == [] and state['on-
       table'] != []:
        return False
    return True
```
In this revised function, we only check the predicates that are present in the goal state. If the 'on' predicate is present, we check that the blocks are stacked in the correct order. If the 'clear' or 'on-table' predicates are present and empty in the goal state, we check that they are also empty in the input state. This ensures that the function returns True for the input state
Goal Iteration 6 - Goal Soundness Test

AutoToS prompt:
The goal test function failed on the following input state {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}, incorrectly reporting it as a non-goal state.
First think step by step what it means for a state to be a goal state in this domain. Then think through in words why the goal test function incorrectly reported input state: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]} as a non-goal state. Now, revise the goal test function and ensure it returns true for the input state.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.
Goal: {'clear': [], 'on-table': [], 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}

Model response:
To be a goal state, the following conditions must be met:
The blocks are stacked in the correct order, as specified in the 'on' predicate.
The 'clear' and 'on-table' predicates are empty, meaning no blocks are clear or on the table.
The goal test function incorrectly reported the input state as a non-goal state because it was checking for exact equality with the goal state, including the 'clear' and 'on-table' predicates. However, the goal state only specifies that these predicates

should be empty, but the input state has blocks 'b' and 'd' that are clear and on the table, respectively.
Here is the revised goal test function:
Python
```
def is_goal_state(state, goal):
    for key, value in goal.items():
        if key == 'on' and set(state['on']) != set(value):
            return False
        elif key in ['clear', 'on-table'] and value == [] and state[
            key] != []:
            return False
    return True
```
In this revised function, we iterate over the goal state's predicates. If the 'on' predicate is present, we check that the blocks are stacked in the correct order. If the 'clear' or 'on-table' predicates are present and empty in the goal state, we check that they are also empty in the input state. This ensures that the function returns True for the input state
Goal Iteration 7 - Goal Soundness Test

AutoToS prompt:
The goal test function failed on the following input state {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}, incorrectly reporting it as a non-goal state.
First think step by step what it means for a state to be a goal state in this domain. Then think through in words why the goal test function incorrectly reported input state: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]} as a non-goal state. Now, revise the goal test function and ensure it returns true for the input state.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.
Goal: {'clear': [], 'on-table': [], 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}

Model response:
To be a goal state, the following conditions must be met:
The blocks are stacked in the correct order, as specified in the 'on' predicate.
The 'clear' and 'on-table' predicates do not need to be empty, but rather, their contents do not matter for the goal state.
The goal test function incorrectly reported the input state as a non-goal state because it was checking for exact equality with the goal state, including the 'clear' and 'on-table' predicates.
Here is the revised goal test function:
Python
```
def is_goal_state(state, goal):
    for key, value in goal.items():
        if key == 'on' and set(state['on']) != set(value):
            return False
    return True
```
In this revised function, we only check the 'on' predicate, ignoring the 'clear' and 'on-table' predicates. This ensures that the function returns True for the input state, as the blocks are stacked in the correct order
Goal Iteration 8 - Goal Soundness Test

AutoToS prompt:
The goal test function failed on the following input state {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on

': [('a', 'c'), ('c', 'd')]}, incorrectly reporting it as a non-goal state.
First think step by step what it means for a state to be a goal state in this domain. Then think through in words why the goal test function incorrectly reported input state: {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]} as a non-goal state. Now, revise the goal test function and ensure it returns true for the input state.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.
Goal: {'clear': [], 'on-table': [], 'on': [('a', 'c')]}

Model response:
To be a goal state, the following conditions must be met:
The blocks are stacked in the correct order, as specified in the 'on' predicate.
The 'clear' and 'on-table' predicates do not need to be empty, but rather, their contents do not matter for the goal state, unless they are specified in the goal.
The goal test function incorrectly reported the input state as a non-goal state because it was checking for exact equality with the goal state, including the 'clear' and 'on-table' predicates.
Here is the revised goal test function:
Python
```
def is_goal_state(state, goal):
    for key, value in goal.items():
        if key == 'on' and not all(pair in state['on'] for pair in
            value):
            return False
        elif key in ['clear', 'on-table'] and value != [] and state[
            key] != value:
            return False
    return True
```
In this revised function, we check the 'on' predicate to ensure all pairs in the goal are present in the state. We also check the 'clear' and 'on-table' predicates only if they are specified in the goal and not empty, to ensure they match the state. This ensures that the function returns True for the input state

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
Input state should not change as a result of applying the successor function.
State before successor function applied: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
State after successor function applied: {'clear': ['a'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
Let's think step by step. First, think of the possible reasons why the input state may change as a result of applying the successor function, such as the use of shallow copy.
Then, provide the complete Python code for the revised successor function that returns a list of successor states and ensure it does not change the input state.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:

The input state may change as a result of applying the successor function due to the use of shallow copy. In Python, when you assign a new variable to an existing list or dictionary, it creates a reference to the original data, not a copy. Therefore, modifying the new variable affects the original data.

Here is the revised successor function that returns a list of successor states without changing the input state:

Python

```
def successor(state):
    successors = []
    state_copy = state.copy()  # Create a deep copy of the state

    if state_copy['arm-empty']:
        # Pick up a block from the table
        for block in state_copy['clear']:
            if block in state_copy['on-table']:
                new_state = state_copy.copy()  # Create a new copy for
                     each successor
                new_state['arm-empty'] = False
                new_state['holding'] = block
                new_state['clear'].remove(block)
                new_state['on-table'].remove(block)
                successors.append(new_state)

        # Unstack a block from on top of another block
        for block1, block2 in state_copy['on']:
            if block1 in state_copy['clear']:
                new_state = state_copy.copy()  # Create a new copy for
                     each successor
                new_state['arm-empty'] = False
                new_state['holding'] = block1
                new_state['clear'].remove(block1)
                new_state['clear'].append(block2)
                successors.append(new_state)

    else:
        # Put down a block on the table
        new_state = state_copy.copy()  # Create a new copy for each
            successor
        new_state['arm-empty'] = True
        new_state['on-table'].append(state_copy['holding'])
        new_state['clear'].append(state_copy['holding'])
        new_state['holding'] = None
        successors.append(new_state)

        # Stack a block on top of another block
        for block in state_copy['clear']:
            new_state = state_copy.copy()  # Create a new copy for
                each successor
            new_state['arm-empty'] = True
            new_state['on'].append((state_copy['holding'], block))
            new_state['clear'].remove(block)
            new_state['holding'] = None
            successors.append(new_state)

    return successors
```

In this revised function, we create a deep copy of the state using the copy() method before generating successor states. We also create a new copy for each successor state to avoid modifying the original state

Successor Iteration 2

AutoToS prompt:

Input state should not change as a result of applying the successor function.
State before successor function applied: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
State after successor function applied: {'clear': ['a'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
Let's think step by step. First, think of the possible reasons why the input state may change as a result of applying the successor function, such as the use of shallow copy.
Then, provide the complete Python code for the revised successor function that returns a list of successor states and ensure it does not change the input state.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The input state may change due to the use of shallow copy or direct modification of the input state.
Here is the revised successor function that returns a list of successor states without changing the input state:
Python
```
import copy

def successor(state):
    successors = []

    # Create a deep copy of the state
    state_copy = copy.deepcopy(state)

    if state_copy['arm-empty']:
        # Pick up a block from the table
        for block in state_copy['on-table']:
            if block in state_copy['clear']:
                new_state = copy.deepcopy(state_copy)
                new_state['arm-empty'] = False
                new_state['holding'] = block
                new_state['clear'].remove(block)
                new_state['on-table'].remove(block)
                successors.append(new_state)

        # Unstack a block from on top of another block
        for block1, block2 in state_copy['on']:
            if block1 in state_copy['clear']:
                new_state = copy.deepcopy(state_copy)
                new_state['arm-empty'] = False
                new_state['holding'] = block1
                new_state['clear'].remove(block1)
                new_state['clear'].append(block2)
                new_state['on'].remove((block1, block2))
                successors.append(new_state)

    else:
        # Put down a block on the table
        new_state = copy.deepcopy(state_copy)
        new_state['arm-empty'] = True
        new_state['on-table'].append(state_copy['holding'])
        new_state['clear'].append(state_copy['holding'])
        new_state['holding'] = None
        successors.append(new_state)

        # Stack a block on top of another block
        for block in state_copy['clear']:
```

```
            new_state = copy.deepcopy(state_copy)
            new_state['arm-empty'] = True
            new_state['on'].append((state_copy['holding'], block))
            new_state['clear'].remove(block)
            new_state['holding'] = None
            successors.append(new_state)

    return successors
```
In this revised function, we use copy.deepcopy() to create deep copies of the state, ensuring that the input state remains unchanged

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 3

AutoToS prompt:
Successor function when run on the state {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]} failed to produce all successors.
Missing successors are: [{'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd'), ('b', 'a')]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The successor function failed to produce all successors because it didn't consider the case where the arm is holding a block and can be stacked on top of another block.
Here is the revised successor function:
Python
```
import copy

def successor(state):
    successors = []

    # Create a deep copy of the state
    state_copy = copy.deepcopy(state)

    if state_copy['arm-empty']:
        # Pick up a block from the table
        for block in state_copy['on-table']:
            if block in state_copy['clear']:
                new_state = copy.deepcopy(state_copy)
                new_state['arm-empty'] = False
                new_state['holding'] = block
                new_state['clear'].remove(block)
                new_state['on-table'].remove(block)
                successors.append(new_state)

        # Unstack a block from on top of another block
        for block1, block2 in state_copy['on']:
            if block1 in state_copy['clear']:
                new_state = copy.deepcopy(state_copy)
                new_state['arm-empty'] = False
                new_state['holding'] = block1
                new_state['clear'].remove(block1)
                new_state['clear'].append(block2)
                new_state['on'].remove((block1, block2))
```

```
                successors.append(new_state)

    else:
        # Put down a block on the table
        new_state = copy.deepcopy(state_copy)
        new_state['arm-empty'] = True
        new_state['on-table'].append(state_copy['holding'])
        new_state['clear'].append(state_copy['holding'])
        new_state['holding'] = None
        successors.append(new_state)

        # Stack a block on top of another block
        for block in state_copy['clear']:
            new_state = copy.deepcopy(state_copy)
            new_state['arm-empty'] = True
            new_state['on'].append((state_copy['holding'], block))
            new_state['clear'].remove(block)
            new_state['holding'] = None
            successors.append(new_state)

        # Stack the held block on top of another block
        for block1, block2 in state_copy['on']:
            if block2 in state_copy['clear']:
                new_state = copy.deepcopy(state_copy)
                new_state['arm-empty'] = True
                new_state['on'].append((state_copy['holding'], block2)
                    )
                new_state['on'].remove((block1, block2))
                new_state['clear'].append(block1)
                new_state['holding'] = None
                successors.append(new_state)

    return successors
```

In this revised function, we added a new case to stack the held block on top of another block, which was missing in the previous implementation

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 4

AutoToS prompt:
Successor function when run on the state {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]} failed to produce all successors.
Missing successors are: [{'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd'), ('b', 'a')]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The successor function failed to produce all successors because it didn't consider the case where the arm is holding a block and can be stacked on top of another block that is clear.
Here is the revised successor function:
Python

```
import copy
```

```
def successor(state):
    successors = []

    # Create a deep copy of the state
    state_copy = copy.deepcopy(state)

    if state_copy[’arm-empty’]:
        # Pick up a block from the table
        for block in state_copy[’on-table’]:
            if block in state_copy[’clear’]:
                new_state = copy.deepcopy(state_copy)
                new_state[’arm-empty’] = False
                new_state[’holding’] = block
                new_state[’clear’].remove(block)
                new_state[’on-table’].remove(block)
                successors.append(new_state)

        # Unstack a block from on top of another block
        for block1, block2 in state_copy[’on’]:
            if block1 in state_copy[’clear’]:
                new_state = copy.deepcopy(state_copy)
                new_state[’arm-empty’] = False
                new_state[’holding’] = block1
                new_state[’clear’].remove(block1)
                new_state[’clear’].append(block2)
                new_state[’on’].remove((block1, block2))
                successors.append(new_state)

    else:
        # Put down a block on the table
        new_state = copy.deepcopy(state_copy)
        new_state[’arm-empty’] = True
        new_state[’on-table’].append(state_copy[’holding’])
        new_state[’clear’].append(state_copy[’holding’])
        new_state[’holding’] = None
        successors.append(new_state)

        # Stack a block on top of another block
        for block in state_copy[’clear’]:
            new_state = copy.deepcopy(state_copy)
            new_state[’arm-empty’] = True
            new_state[’on’].append((state_copy[’holding’], block))
            new_state[’clear’].remove(block)
            new_state[’holding’] = None
            successors.append(new_state)

        # Stack the held block on top of another clear block
        for block in state_copy[’clear’]:
            new_state = copy.deepcopy(state_copy)
            new_state[’arm-empty’] = True
            new_state[’on’].append((state_copy[’holding’], block))
            new_state[’clear’].remove(block)
            new_state[’holding’] = None
            successors.append(new_state)

    return successors
```

In this revised function, we added a new case to stack the held block on top of another clear block, which was missing in the previous implementation

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 5

AutoToS prompt:
Successor function when run on the state {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]} failed to produce all successors.
Missing successors are: [{'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd'), ('b', 'a')]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The successor function failed to produce all successors because it didn't consider the case where the arm is holding a block and can be stacked on top of another clear block.
Here is the revised successor function:
Python
```
import copy

def successor(state):
    successors = []

    # Create a deep copy of the state
    state_copy = copy.deepcopy(state)

    if state_copy['arm-empty']:
        # Pick up a block from the table
        for block in state_copy['on-table']:
            if block in state_copy['clear']:
                new_state = copy.deepcopy(state_copy)
                new_state['arm-empty'] = False
                new_state['holding'] = block
                new_state['clear'].remove(block)
                new_state['on-table'].remove(block)
                successors.append(new_state)

        # Unstack a block from on top of another block
        for block1, block2 in state_copy['on']:
            if block1 in state_copy['clear']:
                new_state = copy.deepcopy(state_copy)
                new_state['arm-empty'] = False
                new_state['holding'] = block1
                new_state['clear'].remove(block1)
                new_state['clear'].append(block2)
                new_state['on'].remove((block1, block2))
                successors.append(new_state)

    else:
        # Put down a block on the table
        new_state = copy.deepcopy(state_copy)
        new_state['arm-empty'] = True
        new_state['on-table'].append(state_copy['holding'])
        new_state['clear'].append(state_copy['holding'])
        new_state['holding'] = None
        successors.append(new_state)

        # Stack a block on top of another block
        for block in state_copy['clear']:
            new_state = copy.deepcopy(state_copy)
            new_state['arm-empty'] = True
            new_state['on'].append((state_copy['holding'], block))
```

```
            new_state['clear'].remove(block)
            new_state['holding'] = None
            successors.append(new_state)

        # Stack the held block on top of another clear block
        for block in state_copy['clear']:
            if block != state_copy['holding']:
                new_state = copy.deepcopy(state_copy)
                new_state['arm-empty'] = True
                new_state['on'].append((state_copy['holding'], block))
                new_state['clear'].remove(block)
                new_state['holding'] = None
                successors.append(new_state)

    return successors
```

```
In this revised function, we added a new case to stack the held block
    on top of another clear block, and also added a condition to avoid
     stacking the held block on top of itself

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 6

AutoToS prompt:
Successor function when run on the state {'clear': ['a'], 'on-table':
    ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c
    ', 'd')]} failed to produce all successors.
Missing successors are: [{'clear': ['b'], 'on-table': ['d'], 'arm-
    empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd'), ('b
    ', 'a')]}]
First think step by step why the successor function failed to produce
    all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.

Model response:
The successor function failed to produce all successors because it
    didn't consider the case where the arm is holding a block and can
    be stacked on top of another clear block, specifically when the
    clear block is not the same as the held block.
Here is the revised successor function:
Python
```

```
import copy

def successor(state):
    successors = []

    # Create a deep copy of the state
    state_copy = copy.deepcopy(state)

    if state_copy['arm-empty']:
        # Pick up a block from the table
        for block in state_copy['on-table']:
            if block in state_copy['clear']:
                new_state = copy.deepcopy(state_copy)
                new_state['arm-empty'] = False
                new_state['holding'] = block
                new_state['clear'].remove(block)
                new_state['on-table'].remove(block)
                successors.append(new_state)
```

```
        # Unstack a block from on top of another block
        for block1, block2 in state_copy['on']:
            if block1 in state_copy['clear']:
                new_state = copy.deepcopy(state_copy)
                new_state['arm-empty'] = False
                new_state['holding'] = block1
                new_state['clear'].remove(block1)
                new_state['clear'].append(block2)
                new_state['on'].remove((block1, block2))
                successors.append(new_state)

    else:
        # Put down a block on the table
        new_state = copy.deepcopy(state_copy)
        new_state['arm-empty'] = True
        new_state['on-table'].append(state_copy['holding'])
        new_state['clear'].append(state_copy['holding'])
        new_state['holding'] = None
        successors.append(new_state)

        # Stack a block on top of another block
        for block in state_copy['clear']:
            if block != state_copy['holding']:
                new_state = copy.deepcopy(state_copy)
                new_state['arm-empty'] = True
                new_state['on'].append((state_copy['holding'], block))
                new_state['clear'].remove(block)
                new_state['holding'] = None
                successors.append(new_state)

        # Stack the held block on top of another clear block
        for block in state_copy['clear']:
            if block != state_copy['holding']:
                new_state = copy.deepcopy(state_copy)
                new_state['arm-empty'] = True
                new_state['on'].append((state_copy['holding'], block))
                new_state['clear'].append(state_copy['holding'])
                new_state['clear'].remove(block)
                new_state['holding'] = None
                successors.append(new_state)

    return successors
```

```
In this revised function, we added a new case to stack the held block
    on top of another clear block, and also added a condition to avoid
     stacking the held block on top of itself. Additionally, we added
    the held block to the clear list when stacking it on top of
    another block

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 502, failure: 0, total expansions 118287, total generated
    281762
```

Listing 33: With partial successor soundness test

```
AutoToS prompt:

I am playing with a set of blocks where I need to arrange the blocks
    into stacks. Here are the actions I can do
   Pick up a block from the table
   Put down a block on the table
```

```
    Unstack a block from on top of another block
    Stack a block on top of another block

    I have the following restrictions on my actions:
    I can only pick up or unstack one block at a time.
    I can only pick up or unstack a block if my hand is empty.
    I can only pick up a block if the block is on the table and the
        block is clear. A block is clear if the block has no other
        blocks on top of it and if the block is not picked up.
    I can only unstack a block from on top of another block if the
        block I am unstacking was really on top of the other block.
    I can only unstack a block from on top of another block if the
        block I am unstacking is clear.
    Once I pick up a block, I am holding the block and it is no longer
        clear and no longer on the table.
    Once I unstack from on top of another block, I am holding the block
         and it is no longer clear. Instead, the other block becomes
        clear.

    I can only put down a block that I am holding.
    I can only stack a block on top of another block if I am holding
        the block being stacked.
    I can only stack a block on top of another block if the block onto
        which I am stacking the block is clear.
    Once I put down a block, my hand becomes empty, the block becomes
        clear, and it is now on the table.
    Once I stack a block on top of another block, the block on top
        becomes clear and the block under it is no longer clear.
If a state is defined by a dictionary of following predicates "clear
    block", "on-table block", "arm-empty", "holding block", and "on
    block1 block2", how are successor states defined?
Here is an example of the initial state:
Init: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding
    ': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
Please provide only the Python code for the successor function that
    returns a list of successor states.

Model response:
Here is the Python code for the successor function:
Python
'''
def successor(state):
    successors = []
    if state['arm-empty']:
        # Pick up a block from the table
        for block in state['clear']:
            if block in state['on-table']:
                new_state = state.copy()
                new_state['clear'].remove(block)
                new_state['on-table'].remove(block)
                new_state['arm-empty'] = False
                new_state['holding'] = block
                successors.append(new_state)

        # Unstack a block from on top of another block
        for block1, block2 in state['on']:
            if block1 in state['clear']:
                new_state = state.copy()
                new_state['clear'].remove(block1)
                new_state['clear'].append(block2)
                new_state['arm-empty'] = False
                new_state['holding'] = block1
                new_state['on'].remove((block1, block2))
                successors.append(new_state)
```

```
    else:
        # Put down a block on the table
        new_state = state.copy()
        new_state['clear'].append(state['holding'])
        new_state['on-table'].append(state['holding'])
        new_state['arm-empty'] = True
        new_state['holding'] = None
        successors.append(new_state)

        # Stack a block on top of another block
        for block in state['clear']:
            new_state = state.copy()
            new_state['clear'].remove(block)
            new_state['clear'].append(state['holding'])
            new_state['on'].append((state['holding'], block))
            new_state['arm-empty'] = True
            new_state['holding'] = None
            successors.append(new_state)

    return successors
```

This function takes a state as input and returns a list of successor states. It checks the current state and applies the possible actions to generate the successor states. Note that this function assumes that the input state is a dictionary with the same structure as the initial state you provided

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state. The goal specifies a partial state, the facts that must be true in any full goal state. Assume that the goal configuration is provided in the same dictionary format.
Here's an example goal specification:
Goal: {'clear': [], 'on-table': [], 'on': [('b', 'c'), ('d', 'b')]}

Model response:
Here is the Python code that tests whether a state is a goal state:
Python

```
def is_goal_state(state, goal):
    for key, value in goal.items():
        if key == 'clear':
            if len(state[key]) != len(value) or any(block not in state
                [key] for block in value):
                return False
        elif key == 'on-table':
            if len(state[key]) != len(value) or any(block not in state
                [key] for block in value):
                return False
        elif key == 'on':
            if len(state[key]) != len(value) or any((block1, block2)
               not in state[key] for block1, block2 in value):
                return False
        elif key == 'arm-empty':
            if state[key] != value:
                return False
        elif key == 'holding':
            if state[key] != value:
                return False
    return True
```

This function takes a state and a goal as input and returns True if the state is a goal state, and False otherwise. It checks each predicate in the goal and verifies that it is true in the state

Goal Iteration 1 - Goal Soundness Test

AutoToS prompt:
The goal test function failed on the following input state {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}, incorrectly reporting it as a non-goal state.
First think step by step what it means for a state to be a goal state in this domain. Then think through in words why the goal test function incorrectly reported input state: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]} as a non-goal state. Now, revise the goal test function and ensure it returns true for the input state.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.
Goal: {'clear': [], 'on-table': [], 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}

Model response:
A state is a goal state if it satisfies all the conditions specified in the goal. In this domain, the goal specifies the desired relationships between blocks (on top of which other blocks) and does not care about the 'clear' and 'on-table' predicates.
The goal test function incorrectly reported the input state as a non-goal state because it checked if the 'clear' and 'on-table' predicates in the state matched the goal, which is not necessary. The goal only cares about the 'on' predicate.
Here is the revised goal test function:
Python
```
def is_goal_state(state, goal):
    for key, value in goal.items():
        if key == 'on':
            if len(state[key]) != len(value) or any((block1, block2)
                not in state[key] for block1, block2 in value):
                return False
        elif key == 'arm-empty':
            if state[key] != value:
                return False
        elif key == 'holding':
            if state[key] != value:
                return False
    return True
```
This revised function only checks the 'on', 'arm-empty', and 'holding' predicates, which are the only ones that matter for the goal in this domain. It will correctly return True for the input state
Goal Iteration 2 - Goal Soundness Test

AutoToS prompt:
The goal test function failed on the following input state {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]}, incorrectly reporting it as a non-goal state.
First think step by step what it means for a state to be a goal state in this domain. Then think through in words why the goal test function incorrectly reported input state: {'clear': ['a'], 'on-table': ['d'], 'arm-empty': False, 'holding': 'b', 'on': [('a', 'c'), ('c', 'd')]} as a non-goal state. Now, revise the goal test function and ensure it returns true for the input state.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.
Goal: {'clear': [], 'on-table': [], 'on': [('a', 'c')]}

Model response:

A state is a goal state if it satisfies all the conditions specified in the goal. In this domain, the goal specifies the desired relationships between blocks (on top of which other blocks) and does not care about the 'clear' and 'on-table' predicates unless they are specified in the goal.
The goal test function incorrectly reported the input state as a non-goal state because it checked if the 'arm-empty' and 'holding' predicates in the state matched the goal, which is not necessary. The goal only cares about the 'on' predicate and the 'clear' and 'on-table' predicates if they are specified.
Here is the revised goal test function:
Python
```
def is_goal_state(state, goal):
    for key, value in goal.items():
        if key == 'on':
            if not all((block1, block2) in state[key] for block1,
                block2 in value):
                return False
        elif key in ['clear', 'on-table']:
            if value and not all(block in state[key] for block in
                value):
                return False
    return True
```
This revised function only checks the predicates that are specified in the goal and ignores the others. It will correctly return True for the input state. Note that it also checks if all the desired 'on' relationships are present in the state, not just if the state's 'on' relationships are a subset of the goal's

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
Input state should not change as a result of applying the successor function.
State before successor function applied: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
State after successor function applied: {'clear': ['a'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd')]}
Let's think step by step. First, think of the possible reasons why the input state may change as a result of applying the successor function, such as the use of shallow copy.
Then, provide the complete Python code for the revised successor function that returns a list of successor states and ensure it does not change the input state.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The input state may change as a result of applying the successor function because the function is modifying the state directly instead of creating a new copy of the state. This is likely due to the use of shallow copy (e.g., new_state = state.copy()) which only creates a new reference to the original state's mutable objects, such as lists.
Here is the revised successor function that returns a list of successor states without modifying the input state:
Python
```
```

```
import copy

def successor(state):
    successors = []
    if state['arm-empty']:
        # Pick up a block from the table
        for block in state['clear']:
            if block in state['on-table']:
                new_state = copy.deepcopy(state)
                new_state['clear'].remove(block)
                new_state['on-table'].remove(block)
                new_state['arm-empty'] = False
                new_state['holding'] = block
                successors.append(new_state)

        # Unstack a block from on top of another block
        for block1, block2 in state['on']:
            if block1 in state['clear']:
                new_state = copy.deepcopy(state)
                new_state['clear'].remove(block1)
                new_state['clear'].append(block2)
                new_state['arm-empty'] = False
                new_state['holding'] = block1
                new_state['on'].remove((block1, block2))
                successors.append(new_state)

    else:
        # Put down a block on the table
        new_state = copy.deepcopy(state)
        new_state['clear'].append(state['holding'])
        new_state['on-table'].append(state['holding'])
        new_state['arm-empty'] = True
        new_state['holding'] = None
        successors.append(new_state)

        # Stack a block on top of another block
        for block in state['clear']:
            new_state = copy.deepcopy(state)
            new_state['clear'].remove(block)
            new_state['clear'].append(state['holding'])
            new_state['on'].append((state['holding'], block))
            new_state['arm-empty'] = True
            new_state['holding'] = None
            successors.append(new_state)

    return successors
```
```
In this revised function, I used copy.deepcopy(state) to create a deep
     copy of the state, which ensures that any modifications made to
    the new state do not affect the original state

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 502, failure: 0, total expansions 50484, total generated
    130127
```

9.2.5 Llama3.1 405B

Listing 34: Without partial successor soundness test

```
AutoToS prompt:
```

I am playing with a set of blocks where I need to arrange the blocks into stacks. Here are the actions I can do
Pick up a block from the table
Put down a block on the table
Unstack a block from on top of another block
Stack a block on top of another block

I have the following restrictions on my actions:
I can only pick up or unstack one block at a time.
I can only pick up or unstack a block if my hand is empty.
I can only pick up a block if the block is on the table and the block is clear. A block is clear if the block has no other blocks on top of it and if the block is not picked up.
I can only unstack a block from on top of another block if the block I am unstacking was really on top of the other block.
I can only unstack a block from on top of another block if the block I am unstacking is clear.
Once I pick up a block, I am holding the block and it is no longer clear and no longer on the table.
Once I unstack from on top of another block, I am holding the block and it is no longer clear. Instead, the other block becomes clear.

I can only put down a block that I am holding.
I can only stack a block on top of another block if I am holding the block being stacked.
I can only stack a block on top of another block if the block onto which I am stacking the block is clear.
Once I put down a block, my hand becomes empty, the block becomes clear, and it is now on the table.
Once I stack a block on top of another block, the block on top becomes clear and the block under it is no longer clear.
If a state is defined by a dictionary of following predicates "clear block", "on-table block", "arm-empty", "holding block", and "on block1 block2", how are successor states defined?
Here is an example of the initial state:
Init: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
Please provide only the Python code for the successor function that returns a list of successor states.

Model response:
Successor Function
Python
```
def successor(state):
    successors = []

    # Pick up a block from the table
    for block in state['on-table']:
        if state['arm-empty'] and block in state['clear']:
            new_state = state.copy()
            new_state['arm-empty'] = False
            new_state['holding'] = block
            new_state['clear'].remove(block)
            new_state['on-table'].remove(block)
            successors.append(new_state)

    # Unstack a block from on top of another block
    for block1, block2 in state['on']:
        if state['arm-empty'] and block1 in state['clear']:
            new_state = state.copy()
            new_state['arm-empty'] = False
            new_state['holding'] = block1
            new_state['clear'].remove(block1)
```

```
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            successors.append(new_state)

    # Put down a block on the table
    if not state['arm-empty']:
        new_state = state.copy()
        new_state['arm-empty'] = True
        new_state['on-table'].append(new_state['holding'])
        new_state['clear'].append(new_state['holding'])
        new_state['holding'] = None
        successors.append(new_state)

    # Stack a block on top of another block
    if not state['arm-empty']:
        for block in state['clear']:
            if block != state['holding']:
                new_state = state.copy()
                new_state['arm-empty'] = True
                new_state['on'].append((new_state['holding'], block))
                new_state['clear'].remove(block)
                new_state['clear'].append(new_state['holding'])
                new_state['holding'] = None
                successors.append(new_state)

    return successors
```

This function generates all possible successor states based on the current state and the actions defined. It iterates over all blocks and checks the preconditions for each action. If the preconditions are met, it creates a new state by applying the effects of the action and adds it to the list of successors.
Generated by Meta Llama 3.1-405

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state. The goal specifies a partial state, the facts that must be true in any full goal state. Assume that the goal configuration is provided in the same dictionary format.
Here's an example goal specification:
Goal: {'clear': [], 'on-table': [], 'on': [('b', 'c'), ('d', 'b')]}

Model response:
Python
```
def is_goal_state(state, goal):
    for predicate, values in goal.items():
        if predicate not in state:
            return False
        if predicate == 'on':
            for block1, block2 in values:
                if (block1, block2) not in state[predicate]:
                    return False
        else:
            for value in values:
                if value not in state[predicate]:
                    return False
    return True
```

This function checks if a given state satisfies the goal conditions by iterating over the predicates and values in the goal specification and verifying if they exist in the state. If any condition is not met, it returns False. If all conditions are met, it returns True.

You can use this function like this:
Python
state = {'clear': ['a'], 'on-table': ['d'], 'arm-empty': True, '
    holding': None, 'on': [('b', 'c'), ('d', 'b'), ('a', 'd')]}
goal = {'clear': [], 'on-table': [], 'on': [('b', 'c'), ('d', 'b')]}
print(is_goal_state(state, goal)) # Output: True
Generated by Meta Llama 3.1-405

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
Input state should not change as a result of applying the successor function.
State before successor function applied: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
State after successor function applied: {'clear': ['a'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd')]}
Let's think step by step. First, think of the possible reasons why the input state may change as a result of applying the successor function, such as the use of shallow copy.
Then, provide the complete Python code for the revised successor function that returns a list of successor states and ensure it does not change the input state.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The issue is due to the use of shallow copy when creating new states in the successor function. In Python, when you use the copy() method or the dict() constructor to copy a dictionary, it only creates a new reference to the same dictionary, not a deep copy. This means that when you modify the new state, you're also modifying the original state.
To fix this, we can use the deepcopy() function from the copy module to create a deep copy of the state dictionary. Here's the revised successor function:
Python
```
import copy

def successor(state):
    successors = []

    # Pick up a block from the table
    for block in state['on-table']:
        if state['arm-empty'] and block in state['clear']:
            new_state = copy.deepcopy(state)
            new_state['arm-empty'] = False
            new_state['holding'] = block
            new_state['clear'].remove(block)
            new_state['on-table'].remove(block)
            successors.append(new_state)

    # Unstack a block from on top of another block
    for block1, block2 in state['on']:
        if state['arm-empty'] and block1 in state['clear']:
            new_state = copy.deepcopy(state)
            new_state['arm-empty'] = False
            new_state['holding'] = block1
            new_state['clear'].remove(block1)
```

```
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            successors.append(new_state)

    # Put down a block on the table
    if not state['arm-empty']:
        new_state = copy.deepcopy(state)
        new_state['arm-empty'] = True
        new_state['on-table'].append(new_state['holding'])
        new_state['clear'].append(new_state['holding'])
        new_state['holding'] = None
        successors.append(new_state)

    # Stack a block on top of another block
    if not state['arm-empty']:
        for block in state['clear']:
            if block != state['holding']:
                new_state = copy.deepcopy(state)
                new_state['arm-empty'] = True
                new_state['on'].append((new_state['holding'], block))
                new_state['clear'].remove(block)
                new_state['clear'].append(new_state['holding'])
                new_state['holding'] = None
                successors.append(new_state)

    return successors
```
```
By using copy.deepcopy(state), we ensure that each new state is a
    completely independent copy of the original state, and modifying
    the new state will not affect the original state.
Generated by Meta Llama 3.1-405

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 502, failure: 0, total expansions 50456, total generated
    130124
```

Listing 35: With partial successor soundness test

```
AutoToS prompt:

I am playing with a set of blocks where I need to arrange the blocks
    into stacks. Here are the actions I can do
   Pick up a block from the table
   Put down a block on the table
   Unstack a block from on top of another block
   Stack a block on top of another block

   I have the following restrictions on my actions:
   I can only pick up or unstack one block at a time.
   I can only pick up or unstack a block if my hand is empty.
   I can only pick up a block if the block is on the table and the
       block is clear. A block is clear if the block has no other
       blocks on top of it and if the block is not picked up.
   I can only unstack a block from on top of another block if the
       block I am unstacking was really on top of the other block.
   I can only unstack a block from on top of another block if the
       block I am unstacking is clear.
   Once I pick up a block, I am holding the block and it is no longer
       clear and no longer on the table.
```

```
   Once I unstack from on top of another block, I am holding the block
       and it is no longer clear. Instead, the other block becomes
      clear.

   I can only put down a block that I am holding.
   I can only stack a block on top of another block if I am holding
      the block being stacked.
   I can only stack a block on top of another block if the block onto
      which I am stacking the block is clear.
   Once I put down a block, my hand becomes empty, the block becomes
      clear, and it is now on the table.
   Once I stack a block on top of another block, the block on top
      becomes clear and the block under it is no longer clear.
If a state is defined by a dictionary of following predicates "clear
   block", "on-table block", "arm-empty", "holding block", and "on
   block1 block2", how are successor states defined?
Here is an example of the initial state:
Init: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding
   ': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
Please provide only the Python code for the successor function that
   returns a list of successor states.

Model response:
Successor Function
Python
```

```
def successor(state):
    """
    Generate a list of successor states based on the current state.

    :param state: A dictionary representing the current state with
        predicates:
                  "clear", "on-table", "arm-empty", "holding", and "on
                      ".
    :return: A list of dictionaries, each representing a possible
        successor state.
    """
    successors = []

    # Pick up a block from the table
    for block in state['on-table']:
        if block in state['clear']:
            new_state = state.copy()
            new_state['holding'] = block
            new_state['arm-empty'] = False
            new_state['clear'].remove(block)
            new_state['on-table'].remove(block)
            successors.append(new_state)

    # Unstack a block from on top of another block
    for block1, block2 in state['on']:
        if block1 in state['clear']:
            new_state = state.copy()
            new_state['holding'] = block1
            new_state['arm-empty'] = False
            new_state['clear'].remove(block1)
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            successors.append(new_state)

    # Put down a block on the table
    if not state['arm-empty']:
        new_state = state.copy()
        new_state['on-table'].append(state['holding'])
        new_state['clear'].append(state['holding'])
```

```
        new_state['holding'] = None
        new_state['arm-empty'] = True
        successors.append(new_state)

    # Stack a block on top of another block
    if not state['arm-empty']:
        for block in state['clear']:
            if block != state['holding']:
                new_state = state.copy()
                new_state['on'].append((state['holding'], block))
                new_state['clear'].remove(block)
                new_state['clear'].append(state['holding'])
                new_state['holding'] = None
                new_state['arm-empty'] = True
                successors.append(new_state)

    return successors
```

You can use this function to generate successor states for a given state. For example:

Python

```
init_state = {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
successors = successor(init_state)
for i, succ in enumerate(successors):
    print(f"Successor {i+1}: {succ}")
```

Generated by Meta Llama 3.1-405

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state. The goal specifies a partial state, the facts that must be true in any full goal state. Assume that the goal configuration is provided in the same dictionary format.
Here's an example goal specification:
Goal: {'clear': [], 'on-table': [], 'on': [('b', 'c'), ('d', 'b')]}

Model response:
Goal Test Function
Python

```
def is_goal(state, goal):
    """
    Check if a state satisfies the goal configuration.

    :param state: A dictionary representing the current state.
    :param goal: A dictionary representing the goal configuration.
    :return: True if the state satisfies the goal, False otherwise.
    """
    for key, value in goal.items():
        if key == 'clear':
            if not all(block not in state['clear'] for block in value):
                return False
        elif key == 'on-table':
            if not all(block not in state['on-table'] for block in value):
                return False
        elif key == 'on':
            if not all(tuple(sorted(on_pair)) in [tuple(sorted(on)) for on in state['on']] for on_pair in value):
                return False
    return True
```

You can use this function to test whether a state is a goal state. For example:

Python

```
state = {'clear': ['a'], 'on-table': ['d'], 'arm-empty': True, '
    holding': None, 'on': [('b', 'c'), ('d', 'b'), ('a', 'd')]}
goal = {'clear': [], 'on-table': [], 'on': [('b', 'c'), ('d', 'b')]}
print(is_goal(state, goal))  # Output: True
```

Generated by Meta Llama 3.1-405

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
Input state should not change as a result of applying the successor function.
State before successor function applied: {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
State after successor function applied: {'clear': ['a'], 'on-table': ['d'], 'arm-empty': True, 'holding': None, 'on': [('a', 'c'), ('c', 'd')]}
Let's think step by step. First, think of the possible reasons why the input state may change as a result of applying the successor function, such as the use of shallow copy.
Then, provide the complete Python code for the revised successor function that returns a list of successor states and ensure it does not change the input state.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
Revised Successor Function
Python
```
import copy

def successor(state):
    """
    Generate a list of successor states based on the current state.

    :param state: A dictionary representing the current state with
        predicates:
                  "clear", "on-table", "arm-empty", "holding", and "on
                      ".
    :return: A list of dictionaries, each representing a possible
        successor state.
    """
    successors = []

    # Pick up a block from the table
    for block in state['on-table']:
        if block in state['clear']:
            new_state = copy.deepcopy(state)
            new_state['holding'] = block
            new_state['arm-empty'] = False
            new_state['clear'].remove(block)
            new_state['on-table'].remove(block)
            successors.append(new_state)

    # Unstack a block from on top of another block
    for block1, block2 in state['on']:
        if block1 in state['clear']:
            new_state = copy.deepcopy(state)
```

```
            new_state['holding'] = block1
            new_state['arm-empty'] = False
            new_state['clear'].remove(block1)
            new_state['clear'].append(block2)
            new_state['on'].remove((block1, block2))
            successors.append(new_state)

    # Put down a block on the table
    if not state['arm-empty']:
        new_state = copy.deepcopy(state)
        new_state['on-table'].append(state['holding'])
        new_state['clear'].append(state['holding'])
        new_state['holding'] = None
        new_state['arm-empty'] = True
        successors.append(new_state)

    # Stack a block on top of another block
    if not state['arm-empty']:
        for block in state['clear']:
            if block != state['holding']:
                new_state = copy.deepcopy(state)
                new_state['on'].append((state['holding'], block))
                new_state['clear'].remove(block)
                new_state['clear'].append(state['holding'])
                new_state['holding'] = None
                new_state['arm-empty'] = True
                successors.append(new_state)

    return successors
```

The main issue was that the state.copy() method creates a shallow copy , which means it only copies the references to the original elements. When we modify the lists in the copied state, it affects the original state as well.

To fix this, we use copy.deepcopy(state) to create a deep copy of the state, which recursively creates new copies of all elements. This ensures that modifying the copied state does not affect the original state.

You can test this revised function with the same example:

Python

```
state = {'clear': ['b'], 'on-table': ['d'], 'arm-empty': True, '
    holding': None, 'on': [('a', 'c'), ('b', 'a'), ('c', 'd')]}
successors = successor(state)
print(state)  # Should remain unchanged
for i, succ in enumerate(successors):
    print(f"Successor {i+1}: {succ}")
```

Generated by Meta Llama 3.1-405

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Failed to validate the plan for problem blocks_instances/instance-0.pddl
['(unstack b a)', '(put-down b)', '(unstack a c)', '(unstack c d)', '(put-down c)', '(pick-up b)', '(stack b d)', '(pick-up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-1.pddl
[]
Failed to validate the plan for problem blocks_instances/instance-10.pddl
[]
Failed to validate the plan for problem blocks_instances/instance-100.pddl

```
['(unstack a c)', '(put-down a)', '(unstack c d)', '(unstack d b)', '(
    put-down d)', '(pick-up b)', '(stack b a)', '(pick-up d)', '(stack
     d b)']
Failed to validate the plan for problem blocks_instances/instance-101.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a b)', '(unstack b d)', '(
    put-down b)', '(pick-up d)', '(stack d c)', '(pick-up b)', '(stack
     b d)']
Failed to validate the plan for problem blocks_instances/instance-103.
    pddl
['(unstack c d)', '(put-down c)', '(unstack d b)', '(put-down d)', '(
    pick-up c)', '(stack c b)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-104.
    pddl
['(unstack a d)', '(put-down a)', '(unstack d b)', '(put-down d)', '(
    pick-up a)', '(stack a d)', '(unstack b c)', '(put-down b)', '(
    pick-up c)', '(stack c a)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-105.
    pddl
['(unstack b a)', '(put-down b)', '(unstack a d)', '(stack a c)', '(
    pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-106.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-107.
    pddl
['(unstack b c)', '(put-down b)', '(unstack d a)', '(pick-up a)', '(
    stack a c)', '(pick-up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-109.
    pddl
['(unstack c a)', '(unstack a d)', '(put-down a)', '(pick-up b)', '(
    stack b d)', '(pick-up a)', '(stack a b)']
Failed to validate the plan for problem blocks_instances/instance-11.
    pddl
['(unstack a d)', '(put-down a)', '(pick-up d)', '(stack d b)', '(pick
    -up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-110.
    pddl
['(unstack a c)', '(put-down a)', '(unstack c d)', '(put-down c)', '(
    unstack d b)', '(stack d a)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-112.
    pddl
['(unstack a d)', '(pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-113.
    pddl
['(unstack b a)', '(put-down b)', '(unstack a d)', '(put-down a)', '(
    unstack d c)', '(stack d a)', '(pick-up c)', '(stack c d)', '(pick
    -up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-114.
    pddl
['(unstack c d)', '(unstack d b)', '(put-down d)', '(unstack b a)', '(
    put-down b)', '(pick-up d)', '(stack d a)', '(pick-up b)', '(stack
     b d)']
Failed to validate the plan for problem blocks_instances/instance-117.
    pddl
['(unstack d b)', '(put-down d)', '(unstack b a)', '(stack b d)', '(
    pick-up c)', '(stack c b)', '(pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-118.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-119.
    pddl
['(unstack c b)', '(put-down c)', '(unstack b a)', '(put-down b)', '(
    unstack a d)', '(stack a c)', '(pick-up d)', '(stack d a)', '(pick
    -up b)', '(stack b d)']
```

```
Failed to validate the plan for problem blocks_instances/instance-12.
    pddl
['(unstack b d)', '(stack b c)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-121.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-122.
    pddl
['(unstack b d)', '(unstack d c)', '(put-down d)', '(unstack c a)', '(
    put-down c)', '(pick-up d)', '(stack d a)', '(pick-up c)', '(stack
     c d)']
Failed to validate the plan for problem blocks_instances/instance-123.
    pddl
['(pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-125.
    pddl
['(unstack a b)', '(stack a d)', '(unstack b c)', '(pick-up c)', '(
    stack c a)']
Failed to validate the plan for problem blocks_instances/instance-126.
    pddl
['(unstack b a)', '(pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-127.
    pddl
['(unstack a c)', '(put-down a)', '(pick-up d)', '(stack d c)', '(pick
    -up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-129.
    pddl
['(unstack d c)', '(put-down d)', '(unstack c b)', '(unstack b a)', '(
    pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-13.
    pddl
['(unstack c b)', '(unstack d a)', '(pick-up a)', '(stack a b)']
Failed to validate the plan for problem blocks_instances/instance-130.
    pddl
['(unstack b a)', '(put-down b)', '(unstack c d)', '(stack c b)', '(
    pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-131.
    pddl
['(unstack a c)', '(unstack c d)', '(put-down c)', '(unstack d b)', '(
    put-down d)', '(pick-up c)', '(stack c b)', '(pick-up d)', '(stack
     d c)']
Failed to validate the plan for problem blocks_instances/instance-133.
    pddl
['(unstack a b)', '(stack a d)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-134.
    pddl
['(unstack d b)', '(unstack b c)', '(put-down b)', '(unstack c a)', '(
    pick-up a)', '(stack a b)']
Failed to validate the plan for problem blocks_instances/instance-136.
    pddl
['(unstack a b)', '(put-down a)', '(unstack d c)', '(pick-up c)', '(
    stack c b)', '(pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-137.
    pddl
['(unstack a c)', '(unstack b d)', '(put-down b)', '(pick-up c)', '(
    stack c d)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-139.
    pddl
['(pick-up a)', '(stack a d)', '(unstack b c)', '(pick-up c)', '(stack
     c a)']
Failed to validate the plan for problem blocks_instances/instance-14.
    pddl
['(unstack a b)', '(put-down a)', '(unstack b c)', '(put-down b)', '(
    unstack c d)', '(stack c b)', '(pick-up d)', '(stack d c)', '(pick
    -up a)', '(stack a d)']
```

```
Failed to validate the plan for problem blocks_instances/instance-141.
    pddl
['(unstack b a)', '(unstack d c)', '(put-down d)', '(pick-up c)', '(
    stack c a)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-143.
    pddl
['(unstack a c)', '(put-down a)', '(pick-up c)', '(stack c d)', '(pick
    -up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-144.
    pddl
['(unstack a b)', '(unstack b d)', '(put-down b)', '(unstack d c)', '(
    pick-up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-145.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a d)', '(stack a b)', '(
    pick-up c)', '(stack c a)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-146.
    pddl
['(unstack c a)', '(put-down c)', '(unstack d b)', '(stack d a)', '(
    pick-up b)', '(stack b d)', '(pick-up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-147.
    pddl
['(unstack a b)', '(put-down a)', '(pick-up d)', '(stack d b)', '(pick
    -up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-148.
    pddl
['(unstack d b)', '(unstack b a)', '(put-down b)', '(unstack a c)', '(
    put-down a)', '(pick-up c)', '(stack c b)', '(pick-up a)', '(stack
     a c)']
Failed to validate the plan for problem blocks_instances/instance-15.
    pddl
['(unstack c b)', '(put-down c)', '(pick-up a)', '(stack a b)', '(pick
    -up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-150.
    pddl
['(unstack b d)', '(unstack d c)', '(put-down d)', '(pick-up c)', '(
    stack c a)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-152.
    pddl
['(unstack d b)', '(put-down d)', '(unstack b c)', '(put-down b)', '(
    unstack c a)', '(stack c d)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-153.
    pddl
['(unstack c d)', '(put-down c)', '(unstack d b)', '(stack d a)', '(
    pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-154.
    pddl
['(unstack c b)', '(put-down c)', '(unstack b a)', '(put-down b)', '(
    unstack a d)', '(stack a c)', '(pick-up b)', '(stack b a)', '(pick
    -up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-155.
    pddl
['(unstack a d)', '(put-down a)', '(unstack d b)', '(unstack b c)', '(
    put-down b)', '(pick-up a)', '(stack a c)', '(pick-up b)', '(stack
     b a)']
Failed to validate the plan for problem blocks_instances/instance-156.
    pddl
['(unstack a d)', '(put-down a)', '(pick-up b)', '(stack b a)', '(
    unstack d c)', '(pick-up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-158.
    pddl
['(pick-up a)', '(stack a c)', '(pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-159.
    pddl
```

```
['(unstack b d)', '(put-down b)', '(pick-up d)', '(stack d a)', '(pick
    -up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-16.
    pddl
['(unstack d c)', '(stack d a)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-160.
    pddl
['(unstack a b)', '(put-down a)', '(unstack b d)', '(unstack d c)', '(
    put-down d)', '(pick-up c)', '(stack c a)', '(pick-up d)', '(stack
     d c)']
Failed to validate the plan for problem blocks_instances/instance-162.
    pddl
['(unstack a d)', '(stack a b)', '(pick-up d)', '(stack d a)']
Failed to validate the plan for problem blocks_instances/instance-163.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a d)', '(pick-up c)', '(
    stack c d)']
Failed to validate the plan for problem blocks_instances/instance-164.
    pddl
['(pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-165.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-168.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a b)', '(stack a d)', '(
    pick-up c)', '(stack c a)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-17.
    pddl
['(unstack a d)', '(unstack d c)', '(put-down d)', '(pick-up c)', '(
    stack c b)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-171.
    pddl
['(unstack d b)', '(put-down d)', '(unstack b c)', '(stack b d)', '(
    pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-172.
    pddl
['(unstack a c)', '(put-down a)', '(unstack c d)', '(pick-up d)', '(
    stack d a)']
Failed to validate the plan for problem blocks_instances/instance-174.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a b)', '(put-down a)', '(
    pick-up b)', '(stack b c)', '(pick-up d)', '(stack d b)', '(pick-
    up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-175.
    pddl
['(unstack c a)', '(unstack d b)', '(pick-up a)', '(stack a b)']
Failed to validate the plan for problem blocks_instances/instance-176.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-18.
    pddl
['(unstack d b)', '(put-down d)', '(unstack b a)', '(put-down b)', '(
    unstack a c)', '(pick-up c)', '(stack c d)', '(pick-up b)', '(
    stack b c)']
Failed to validate the plan for problem blocks_instances/instance-180.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-181.
    pddl
['(unstack b a)', '(put-down b)', '(pick-up d)', '(stack d a)', '(pick
    -up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-182.
    pddl
```

```
['(unstack d b)', '(put-down d)', '(unstack b a)', '(put-down b)', '(
    unstack a c)', '(stack a d)', '(pick-up b)', '(stack b a)', '(pick
    -up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-183.
    pddl
['(unstack b c)', '(unstack c d)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-184.
    pddl
['(unstack d b)', '(pick-up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-186.
    pddl
['(unstack b c)', '(put-down b)', '(unstack c d)', '(pick-up b)', '(
    stack b d)']
Failed to validate the plan for problem blocks_instances/instance-187.
    pddl
['(unstack b d)', '(put-down b)', '(pick-up c)', '(stack c b)', '(pick
    -up d)', '(stack d c)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-188.
    pddl
['(unstack b c)', '(unstack c a)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-189.
    pddl
['(unstack c b)', '(put-down c)', '(unstack b a)', '(put-down b)', '(
    unstack a d)', '(stack a c)', '(pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-190.
    pddl
['(unstack b a)', '(put-down b)', '(pick-up a)', '(stack a d)', '(pick
    -up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-192.
    pddl
['(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-194.
    pddl
['(unstack b a)', '(stack b d)', '(unstack a c)', '(pick-up c)', '(
    stack c b)']
Failed to validate the plan for problem blocks_instances/instance-196.
    pddl
['(unstack b d)', '(unstack d c)', '(put-down d)', '(pick-up a)', '(
    stack a c)', '(pick-up d)', '(stack d a)']
Failed to validate the plan for problem blocks_instances/instance-197.
    pddl
['(unstack c b)', '(put-down c)', '(pick-up a)', '(stack a c)', '(
    unstack b d)', '(pick-up d)', '(stack d a)']
Failed to validate the plan for problem blocks_instances/instance-198.
    pddl
['(unstack a b)', '(pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-199.
    pddl
['(unstack b d)', '(unstack d a)', '(put-down d)', '(unstack a c)', '(
    pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-2.
    pddl
['(unstack d c)', '(pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-200.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a b)', '(put-down a)', '(
    pick-up d)', '(stack d a)', '(unstack b c)', '(put-down b)', '(
    pick-up c)', '(stack c d)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-201.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-202.
    pddl
[]
```

```
Failed to validate the plan for problem blocks_instances/instance-203.
    pddl
['(unstack d c)', '(pick-up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-206.
    pddl
['(pick-up a)', '(stack a b)', '(pick-up d)', '(stack d a)']
Failed to validate the plan for problem blocks_instances/instance-208.
    pddl
['(unstack d c)', '(put-down d)', '(unstack c a)', '(pick-up d)', '(
    stack d a)']
Failed to validate the plan for problem blocks_instances/instance-209.
    pddl
['(unstack c b)', '(put-down c)', '(unstack b a)', '(unstack a d)', '(
    put-down a)', '(pick-up d)', '(stack d c)', '(pick-up a)', '(stack
     a d)']
Failed to validate the plan for problem blocks_instances/instance-210.
    pddl
['(unstack a d)', '(pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-211.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-212.
    pddl
['(unstack c b)', '(put-down c)', '(pick-up d)', '(stack d b)', '(pick
    -up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-213.
    pddl
['(unstack d b)', '(put-down d)', '(unstack b c)', '(put-down b)', '(
    unstack c a)', '(stack c d)', '(pick-up b)', '(stack b c)', '(pick
    -up a)', '(stack a b)']
Failed to validate the plan for problem blocks_instances/instance-214.
    pddl
['(pick-up c)', '(stack c b)', '(unstack a d)', '(put-down a)', '(pick
    -up d)', '(stack d c)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-216.
    pddl
['(unstack d b)', '(pick-up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-217.
    pddl
['(unstack a c)', '(put-down a)', '(unstack c d)', '(stack c a)', '(
    pick-up d)', '(stack d c)', '(pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-218.
    pddl
['(unstack a c)', '(put-down a)', '(unstack b d)', '(stack b c)', '(
    pick-up d)', '(stack d a)']
Failed to validate the plan for problem blocks_instances/instance-22.
    pddl
['(unstack d c)', '(put-down d)', '(unstack c a)', '(stack c b)', '(
    pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-220.
    pddl
['(unstack c b)', '(put-down c)', '(unstack b a)', '(stack b d)', '(
    pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-222.
    pddl
['(unstack d b)', '(put-down d)', '(unstack a c)', '(stack a d)', '(
    pick-up b)', '(stack b a)', '(pick-up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-225.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-227.
    pddl
['(unstack a c)', '(put-down a)', '(pick-up b)', '(stack b a)', '(pick
    -up d)', '(stack d b)', '(pick-up c)', '(stack c d)']
```

```
Failed to validate the plan for problem blocks_instances/instance-228.
    pddl
['(unstack b d)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-229.
    pddl
['(unstack d b)', '(put-down d)', '(unstack b c)', '(pick-up c)', '(
    stack c d)']
Failed to validate the plan for problem blocks_instances/instance-230.
    pddl
['(unstack c d)', '(put-down c)', '(unstack d b)', '(put-down d)', '(
    unstack b a)', '(stack b c)', '(pick-up d)', '(stack d b)', '(pick
    -up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-232.
    pddl
['(pick-up c)', '(stack c b)', '(unstack d a)', '(pick-up a)', '(stack
     a c)']
Failed to validate the plan for problem blocks_instances/instance-234.
    pddl
['(unstack a d)', '(put-down a)', '(unstack d b)', '(pick-up a)', '(
    stack a b)']
Failed to validate the plan for problem blocks_instances/instance-235.
    pddl
['(unstack d b)', '(put-down d)', '(pick-up b)', '(stack b a)', '(pick
    -up c)', '(stack c b)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-237.
    pddl
['(unstack b a)', '(put-down b)', '(pick-up a)', '(stack a c)', '(pick
    -up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-238.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a c)', '(stack a d)', '(
    pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-239.
    pddl
['(unstack c a)', '(stack c d)', '(pick-up a)', '(stack a c)', '(pick-
    up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-240.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-241.
    pddl
['(unstack c b)', '(put-down c)', '(unstack a d)', '(stack a b)', '(
    pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-242.
    pddl
['(unstack b a)', '(put-down b)', '(unstack a d)', '(put-down a)', '(
    pick-up b)', '(stack b a)', '(unstack d c)', '(pick-up c)', '(
    stack c b)']
Failed to validate the plan for problem blocks_instances/instance-243.
    pddl
['(unstack b a)', '(put-down b)', '(unstack a c)', '(stack a d)', '(
    pick-up c)', '(stack c a)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-244.
    pddl
['(unstack c d)', '(put-down c)', '(unstack d b)', '(stack d c)', '(
    unstack b a)', '(put-down b)', '(pick-up a)', '(stack a d)', '(
    pick-up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-245.
    pddl
['(unstack b c)', '(put-down b)', '(unstack d a)', '(stack d c)', '(
    pick-up a)', '(stack a b)']
Failed to validate the plan for problem blocks_instances/instance-246.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a b)', '(stack a c)', '(
    pick-up b)', '(stack b a)', '(pick-up d)', '(stack d b)']
```

```
Failed to validate the plan for problem blocks_instances/instance-247.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a d)', '(unstack d b)', '(
    pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-248.
    pddl
['(pick-up a)', '(stack a c)', '(pick-up b)', '(stack b a)', '(pick-up
     d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-249.
    pddl
['(unstack d a)', '(put-down d)', '(unstack c b)', '(stack c a)', '(
    pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-25.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-250.
    pddl
['(unstack b c)', '(put-down b)', '(unstack c d)', '(stack c b)', '(
    unstack d a)', '(pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-251.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a d)', '(put-down a)', '(
    unstack d b)', '(put-down d)', '(pick-up c)', '(stack c d)', '(
    pick-up a)', '(stack a c)', '(pick-up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-254.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a b)', '(stack a c)', '(
    pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-255.
    pddl
['(unstack a b)', '(put-down a)', '(unstack c d)', '(stack c b)', '(
    pick-up d)', '(stack d c)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-257.
    pddl
['(unstack a c)', '(put-down a)', '(unstack b d)', '(put-down b)', '(
    pick-up c)', '(stack c b)', '(pick-up d)', '(stack d c)', '(pick-
    up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-258.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a c)', '(put-down a)', '(
    pick-up d)', '(stack d c)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-259.
    pddl
['(unstack b c)', '(stack b a)', '(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-260.
    pddl
['(unstack a b)', '(put-down a)', '(unstack b c)', '(stack b d)', '(
    pick-up a)', '(stack a b)']
Failed to validate the plan for problem blocks_instances/instance-261.
    pddl
['(unstack c a)', '(unstack a d)', '(pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-262.
    pddl
['(unstack d a)', '(put-down d)', '(pick-up a)', '(stack a b)', '(pick
    -up c)', '(stack c a)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-264.
    pddl
['(unstack b d)', '(put-down b)', '(unstack d c)', '(stack d b)', '(
    pick-up a)', '(stack a d)', '(pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-265.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-266.
    pddl
```

```
['(unstack a b)', '(stack a d)', '(pick-up b)', '(stack b a)', '(pick-
    up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-27.
    pddl
['(unstack b d)', '(unstack c a)', '(put-down c)', '(pick-up a)', '(
    stack a d)', '(pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-270.
    pddl
['(unstack a c)', '(put-down a)', '(unstack c d)', '(put-down c)', '(
    unstack d b)', '(stack d c)', '(pick-up a)', '(stack a d)', '(pick
    -up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-273.
    pddl
['(pick-up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-274.
    pddl
['(unstack d a)', '(stack d c)', '(pick-up a)', '(stack a b)']
Failed to validate the plan for problem blocks_instances/instance-275.
    pddl
['(unstack b d)', '(stack b a)', '(pick-up c)', '(stack c b)', '(pick-
    up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-276.
    pddl
['(unstack a b)', '(put-down a)', '(unstack b c)', '(put-down b)', '(
    pick-up d)', '(stack d b)', '(pick-up a)', '(stack a d)', '(pick-
    up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-280.
    pddl
['(unstack a c)', '(put-down a)', '(unstack c d)', '(put-down c)', '(
    pick-up a)', '(stack a c)', '(unstack d b)', '(put-down d)', '(
    pick-up b)', '(stack b a)', '(pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-281.
    pddl
['(unstack b c)', '(put-down b)', '(pick-up d)', '(stack d b)', '(
    unstack c a)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-282.
    pddl
['(unstack c b)', '(put-down c)', '(unstack b a)', '(put-down b)', '(
    pick-up c)', '(stack c b)', '(unstack a d)', '(put-down a)', '(
    pick-up d)', '(stack d c)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-284.
    pddl
['(unstack b d)', '(put-down b)', '(unstack d c)', '(stack d b)', '(
    pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-287.
    pddl
['(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-29.
    pddl
['(unstack a d)', '(put-down a)', '(unstack c b)', '(stack c d)', '(
    pick-up b)', '(stack b c)', '(pick-up a)', '(stack a b)']
Failed to validate the plan for problem blocks_instances/instance-292.
    pddl
['(unstack a d)', '(put-down a)', '(unstack d b)', '(pick-up a)', '(
    stack a b)']
Failed to validate the plan for problem blocks_instances/instance-293.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-294.
    pddl
['(unstack c d)', '(put-down c)', '(unstack d a)', '(unstack a b)', '(
    put-down a)', '(pick-up c)', '(stack c b)', '(pick-up a)', '(stack
     a c)']
Failed to validate the plan for problem blocks_instances/instance-295.
    pddl
```

```
['(unstack d b)', '(put-down d)', '(unstack c a)', '(stack c b)', '(
    pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-296.
    pddl
['(unstack a c)', '(unstack c d)', '(put-down c)', '(pick-up d)', '(
    stack d b)', '(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-297.
    pddl
['(unstack d a)', '(pick-up a)', '(stack a b)']
Failed to validate the plan for problem blocks_instances/instance-298.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a d)', '(put-down a)', '(
    pick-up d)', '(stack d c)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-3.
    pddl
['(unstack b c)', '(unstack c d)', '(put-down c)', '(unstack d a)', '(
    put-down d)', '(pick-up a)', '(stack a c)', '(pick-up d)', '(stack
     d a)']
Failed to validate the plan for problem blocks_instances/instance-30.
    pddl
['(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-300.
    pddl
['(unstack b d)', '(put-down b)', '(pick-up d)', '(stack d c)', '(pick
    -up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-301.
    pddl
['(unstack c a)', '(stack c b)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-302.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a b)', '(put-down a)', '(
    pick-up b)', '(stack b d)', '(pick-up c)', '(stack c b)', '(pick-
    up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-304.
    pddl
['(unstack b c)', '(put-down b)', '(unstack c a)', '(put-down c)', '(
    pick-up b)', '(stack b c)', '(unstack a d)', '(pick-up d)', '(
    stack d b)']
Failed to validate the plan for problem blocks_instances/instance-305.
    pddl
['(unstack d c)', '(put-down d)', '(unstack c b)', '(pick-up b)', '(
    stack b a)', '(pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-306.
    pddl
['(pick-up a)', '(stack a d)', '(unstack b c)', '(pick-up c)', '(stack
     c a)']
Failed to validate the plan for problem blocks_instances/instance-307.
    pddl
['(unstack a c)', '(unstack c d)', '(put-down c)', '(unstack d b)', '(
    put-down d)', '(pick-up c)', '(stack c b)', '(pick-up d)', '(stack
     d c)']
Failed to validate the plan for problem blocks_instances/instance-309.
    pddl
['(unstack a d)', '(put-down a)', '(unstack d c)', '(put-down d)', '(
    unstack c b)', '(stack c a)', '(pick-up b)', '(stack b c)', '(pick
    -up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-310.
    pddl
['(unstack c a)', '(put-down c)', '(pick-up a)', '(stack a d)', '(pick
    -up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-311.
    pddl
['(unstack d b)', '(put-down d)', '(unstack b c)', '(put-down b)', '(
    pick-up c)', '(stack c d)', '(pick-up b)', '(stack b c)']
```

```
Failed to validate the plan for problem blocks_instances/instance-312.
    pddl
['(pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-313.
    pddl
['(pick-up a)', '(stack a b)', '(unstack d c)', '(pick-up c)', '(stack
     c a)']
Failed to validate the plan for problem blocks_instances/instance-315.
    pddl
['(unstack a c)', '(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-316.
    pddl
['(unstack b d)', '(put-down b)', '(pick-up a)', '(stack a b)', '(pick
    -up c)', '(stack c a)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-320.
    pddl
['(unstack b d)', '(put-down b)', '(unstack a c)', '(stack a b)', '(
    pick-up c)', '(stack c a)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-322.
    pddl
['(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-327.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-330.
    pddl
['(unstack b d)', '(put-down b)', '(unstack d a)', '(put-down d)', '(
    unstack a c)', '(put-down a)', '(pick-up b)', '(stack b c)', '(
    pick-up d)', '(stack d b)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-331.
    pddl
['(unstack d b)', '(put-down d)', '(unstack a c)', '(stack a b)', '(
    pick-up d)', '(stack d a)']
Failed to validate the plan for problem blocks_instances/instance-332.
    pddl
['(unstack d b)', '(put-down d)', '(unstack b a)', '(put-down b)', '(
    unstack a c)', '(stack a d)', '(pick-up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-333.
    pddl
['(unstack d b)', '(stack d a)', '(pick-up b)', '(stack b d)', '(pick-
    up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-334.
    pddl
['(pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-337.
    pddl
['(unstack c d)', '(put-down c)', '(pick-up b)', '(stack b c)', '(
    unstack d a)', '(put-down d)', '(pick-up a)', '(stack a b)', '(
    pick-up d)', '(stack d a)']
Failed to validate the plan for problem blocks_instances/instance-340.
    pddl
['(unstack d c)', '(put-down d)', '(unstack c b)', '(stack c d)', '(
    unstack b a)', '(pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-342.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a c)', '(put-down a)', '(
    unstack c b)', '(put-down c)', '(pick-up b)', '(stack b d)', '(
    pick-up a)', '(stack a b)', '(pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-343.
    pddl
['(unstack b a)', '(unstack a d)', '(put-down a)', '(unstack d c)', '(
    put-down d)', '(pick-up c)', '(stack c a)', '(pick-up d)', '(stack
     d c)']
Failed to validate the plan for problem blocks_instances/instance-344.
    pddl
```

```
['(unstack b d)', '(put-down b)', '(unstack c a)', '(stack c b)', '(
    pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-346.
    pddl
['(unstack a c)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-347.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-35.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a d)', '(pick-up c)', '(
    stack c d)']
Failed to validate the plan for problem blocks_instances/instance-350.
    pddl
['(unstack b d)', '(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-353.
    pddl
['(pick-up b)', '(stack b a)', '(pick-up c)', '(stack c b)', '(pick-up
     d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-354.
    pddl
['(unstack a d)', '(put-down a)', '(unstack d c)', '(put-down d)', '(
    unstack c b)', '(put-down c)', '(pick-up b)', '(stack b a)', '(
    pick-up d)', '(stack d b)', '(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-355.
    pddl
['(unstack c a)', '(put-down c)', '(pick-up b)', '(stack b c)', '(
    unstack a d)', '(pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-356.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a c)', '(stack a d)', '(
    pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-357.
    pddl
['(unstack b c)', '(put-down b)', '(unstack d a)', '(pick-up a)', '(
    stack a c)', '(pick-up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-358.
    pddl
['(unstack a d)', '(put-down a)', '(unstack d c)', '(unstack c b)', '(
    put-down c)', '(pick-up a)', '(stack a b)', '(pick-up c)', '(stack
     c a)']
Failed to validate the plan for problem blocks_instances/instance-359.
    pddl
['(unstack c b)', '(stack c a)', '(unstack b d)', '(pick-up d)', '(
    stack d c)']
Failed to validate the plan for problem blocks_instances/instance-36.
    pddl
['(unstack b d)', '(stack b a)', '(unstack d c)', '(pick-up c)', '(
    stack c b)']
Failed to validate the plan for problem blocks_instances/instance-360.
    pddl
['(unstack a b)', '(put-down a)', '(unstack d c)', '(stack d b)', '(
    pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-361.
    pddl
['(unstack c d)', '(put-down c)', '(unstack d b)', '(put-down d)', '(
    unstack b a)', '(stack b d)', '(pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-363.
    pddl
['(unstack a d)', '(put-down a)', '(unstack d b)', '(stack d a)', '(
    pick-up b)', '(stack b d)', '(pick-up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-364.
    pddl
['(unstack a b)', '(put-down a)', '(unstack b d)', '(pick-up a)', '(
    stack a d)']
```

```
Failed to validate the plan for problem blocks_instances/instance-365.
    pddl
['(unstack d c)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-366.
    pddl
['(pick-up b)', '(stack b d)', '(unstack a c)', '(pick-up c)', '(stack
     c b)']
Failed to validate the plan for problem blocks_instances/instance-367.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a d)', '(put-down a)', '(
    unstack d b)', '(pick-up b)', '(stack b c)', '(pick-up a)', '(
    stack a b)']
Failed to validate the plan for problem blocks_instances/instance-368.
    pddl
['(pick-up c)', '(stack c a)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-369.
    pddl
['(unstack a d)', '(put-down a)', '(unstack d b)', '(pick-up a)', '(
    stack a b)']
Failed to validate the plan for problem blocks_instances/instance-37.
    pddl
['(unstack b d)', '(put-down b)', '(unstack d a)', '(stack d c)', '(
    pick-up a)', '(stack a b)']
Failed to validate the plan for problem blocks_instances/instance-370.
    pddl
['(unstack c d)', '(put-down c)', '(unstack d a)', '(stack d c)', '(
    unstack a b)', '(pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-371.
    pddl
['(unstack a b)', '(put-down a)', '(unstack b d)', '(put-down b)', '(
    pick-up a)', '(stack a b)', '(unstack d c)', '(pick-up c)', '(
    stack c a)']
Failed to validate the plan for problem blocks_instances/instance-373.
    pddl
['(unstack b a)', '(put-down b)', '(unstack a c)', '(put-down a)', '(
    pick-up b)', '(stack b a)', '(unstack c d)', '(pick-up d)', '(
    stack d b)']
Failed to validate the plan for problem blocks_instances/instance-374.
    pddl
['(unstack b c)', '(unstack d a)', '(put-down d)', '(pick-up c)', '(
    stack c a)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-375.
    pddl
['(unstack a b)', '(put-down a)', '(unstack b d)', '(put-down b)', '(
    unstack d c)', '(stack d a)', '(pick-up b)', '(stack b d)', '(pick
    -up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-376.
    pddl
['(unstack a d)', '(put-down a)', '(unstack d b)', '(stack d c)', '(
    pick-up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-38.
    pddl
['(unstack a d)', '(put-down a)', '(unstack d b)', '(unstack b c)', '(
    put-down b)', '(pick-up a)', '(stack a c)', '(pick-up b)', '(stack
     b a)']
Failed to validate the plan for problem blocks_instances/instance-380.
    pddl
['(unstack c d)', '(put-down c)', '(unstack d b)', '(unstack b a)', '(
    pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-381.
    pddl
['(unstack a d)', '(put-down a)', '(pick-up b)', '(stack b a)', '(pick
    -up c)', '(stack c b)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-382.
    pddl
```

```
['(unstack c a)', '(stack c b)', '(unstack a d)', '(pick-up d)', '(
    stack d c)']
Failed to validate the plan for problem blocks_instances/instance-383.
    pddl
['(unstack a b)', '(put-down a)', '(unstack b c)', '(put-down b)', '(
    pick-up c)', '(stack c a)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-384.
    pddl
['(unstack d b)', '(stack d a)', '(pick-up c)', '(stack c d)', '(pick-
    up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-385.
    pddl
['(unstack d c)', '(put-down d)', '(unstack c b)', '(stack c a)', '(
    pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-386.
    pddl
['(unstack c d)', '(put-down c)', '(unstack d b)', '(unstack b a)', '(
    put-down b)', '(pick-up a)', '(stack a c)', '(pick-up b)', '(stack
     b a)']
Failed to validate the plan for problem blocks_instances/instance-387.
    pddl
['(unstack b c)', '(put-down b)', '(unstack c a)', '(pick-up b)', '(
    stack b a)']
Failed to validate the plan for problem blocks_instances/instance-388.
    pddl
['(unstack b d)', '(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-391.
    pddl
['(unstack b a)', '(stack b c)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-393.
    pddl
['(unstack a b)', '(unstack b d)', '(put-down b)', '(unstack d c)', '(
    put-down d)', '(pick-up c)', '(stack c b)', '(pick-up d)', '(stack
     d c)']
Failed to validate the plan for problem blocks_instances/instance-394.
    pddl
['(unstack c b)', '(pick-up b)', '(stack b a)', '(pick-up d)', '(stack
     d b)']
Failed to validate the plan for problem blocks_instances/instance-395.
    pddl
['(pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-396.
    pddl
['(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-397.
    pddl
['(unstack c b)', '(unstack b a)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-398.
    pddl
['(unstack a c)', '(put-down a)', '(pick-up d)', '(stack d a)', '(
    unstack c b)', '(put-down c)', '(pick-up b)', '(stack b d)', '(
    pick-up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-399.
    pddl
['(unstack b d)', '(put-down b)', '(unstack d c)', '(put-down d)', '(
    pick-up b)', '(stack b d)', '(unstack c a)', '(put-down c)', '(
    pick-up a)', '(stack a b)', '(pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-4.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a c)', '(put-down a)', '(
    pick-up d)', '(stack d a)', '(unstack c b)', '(pick-up b)', '(
    stack b d)']
Failed to validate the plan for problem blocks_instances/instance-40.
    pddl
```

```
['(unstack c d)', '(put-down c)', '(unstack d a)', '(put-down d)', '(
    unstack a b)', '(put-down a)', '(pick-up c)', '(stack c a)', '(
    pick-up d)', '(stack d c)', '(pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-400.
    pddl
['(unstack d b)', '(put-down d)', '(unstack b c)', '(stack b d)', '(
    pick-up c)', '(stack c b)', '(pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-401.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-402.
    pddl
['(unstack c d)', '(unstack d b)', '(put-down d)', '(unstack b a)', '(
    put-down b)', '(pick-up a)', '(stack a d)', '(pick-up b)', '(stack
     b a)']
Failed to validate the plan for problem blocks_instances/instance-403.
    pddl
['(pick-up a)', '(stack a d)', '(pick-up b)', '(stack b a)', '(pick-up
     c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-404.
    pddl
['(unstack a c)', '(stack a b)', '(pick-up d)', '(stack d a)']
Failed to validate the plan for problem blocks_instances/instance-405.
    pddl
['(unstack c d)', '(put-down c)', '(pick-up b)', '(stack b c)', '(
    unstack d a)', '(pick-up a)', '(stack a b)']
Failed to validate the plan for problem blocks_instances/instance-406.
    pddl
['(unstack c b)', '(put-down c)', '(unstack b a)', '(put-down b)', '(
    unstack a d)', '(stack a c)', '(pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-408.
    pddl
['(unstack b d)', '(put-down b)', '(unstack d a)', '(put-down d)', '(
    unstack a c)', '(stack a b)', '(pick-up c)', '(stack c a)', '(pick
    -up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-409.
    pddl
['(unstack d c)', '(put-down d)', '(pick-up c)', '(stack c b)', '(pick
    -up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-41.
    pddl
['(pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-411.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-412.
    pddl
['(unstack a b)', '(put-down a)', '(unstack b d)', '(put-down b)', '(
    pick-up a)', '(stack a b)', '(unstack d c)', '(pick-up c)', '(
    stack c a)']
Failed to validate the plan for problem blocks_instances/instance-413.
    pddl
['(pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-414.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-415.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a b)', '(stack a c)', '(
    pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-416.
    pddl
['(unstack a b)', '(put-down a)', '(unstack b c)', '(put-down b)', '(
    unstack c d)', '(stack c a)', '(pick-up b)', '(stack b c)']
```

```
Failed to validate the plan for problem blocks_instances/instance-417.
    pddl
['(unstack b c)', '(unstack c a)', '(put-down c)', '(unstack a d)', '(
    pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-418.
    pddl
['(unstack b c)', '(put-down b)', '(unstack c d)', '(unstack d a)', '(
    put-down d)', '(pick-up b)', '(stack b a)', '(pick-up d)', '(stack
     d b)']
Failed to validate the plan for problem blocks_instances/instance-420.
    pddl
['(unstack d c)', '(put-down d)', '(unstack c b)', '(put-down c)', '(
    unstack b a)', '(stack b d)', '(pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-422.
    pddl
['(pick-up a)', '(stack a d)', '(unstack c b)', '(pick-up b)', '(stack
     b a)']
Failed to validate the plan for problem blocks_instances/instance-423.
    pddl
['(pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-424.
    pddl
['(unstack d c)', '(put-down d)', '(unstack c b)', '(stack c a)', '(
    pick-up d)', '(stack d c)', '(pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-425.
    pddl
['(unstack a c)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-427.
    pddl
['(unstack b c)', '(put-down b)', '(unstack c d)', '(put-down c)', '(
    unstack d a)', '(stack d b)', '(pick-up c)', '(stack c d)', '(pick
    -up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-429.
    pddl
['(unstack d c)', '(put-down d)', '(unstack c b)', '(put-down c)', '(
    unstack b a)', '(stack b c)', '(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-43.
    pddl
['(unstack d a)', '(unstack a b)', '(put-down a)', '(pick-up c)', '(
    stack c b)', '(pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-430.
    pddl
['(unstack a b)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-431.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-433.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-434.
    pddl
['(unstack a c)', '(put-down a)', '(pick-up c)', '(stack c d)', '(pick
    -up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-435.
    pddl
['(unstack a b)', '(put-down a)', '(unstack b c)', '(unstack c d)', '(
    put-down c)', '(pick-up a)', '(stack a d)', '(pick-up c)', '(stack
     c a)']
Failed to validate the plan for problem blocks_instances/instance-436.
    pddl
['(unstack c a)', '(pick-up a)', '(stack a b)']
Failed to validate the plan for problem blocks_instances/instance-437.
    pddl
```

```
['(unstack a b)', '(put-down a)', '(pick-up b)', '(stack b a)', '(
    unstack d c)', '(put-down d)', '(pick-up c)', '(stack c b)', '(
    pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-438.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-44.
    pddl
['(unstack a c)', '(put-down a)', '(unstack c b)', '(unstack b d)', '(
    put-down b)', '(pick-up a)', '(stack a d)', '(pick-up b)', '(stack
     b a)']
Failed to validate the plan for problem blocks_instances/instance-440.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a d)', '(stack a c)', '(
    pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-441.
    pddl
['(unstack c b)', '(put-down c)', '(pick-up b)', '(stack b a)', '(pick
    -up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-443.
    pddl
['(unstack b a)', '(put-down b)', '(unstack d c)', '(stack d a)', '(
    pick-up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-444.
    pddl
['(unstack c d)', '(pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-445.
    pddl
['(unstack d b)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-446.
    pddl
['(unstack a e)', '(unstack e c)', '(put-down e)', '(unstack c b)', '(
    put-down c)', '(unstack b d)', '(pick-up d)', '(stack d e)', '(
    pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-447.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a d)', '(stack a c)', '(
    unstack b e)', '(stack b a)', '(pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-451.
    pddl
['(pick-up e)', '(stack e c)', '(unstack b a)', '(stack b e)', '(pick-
    up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-452.
    pddl
['(unstack d a)', '(unstack a e)', '(put-down a)', '(unstack e c)', '(
    stack e a)', '(unstack c b)', '(put-down c)', '(pick-up b)', '(
    stack b e)', '(pick-up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-453.
    pddl
['(unstack e a)', '(put-down e)', '(unstack a d)', '(unstack d b)', '(
    put-down d)', '(unstack b c)', '(stack b e)', '(pick-up d)', '(
    stack d b)']
Failed to validate the plan for problem blocks_instances/instance-454.
    pddl
['(unstack b c)', '(put-down b)', '(unstack c d)', '(unstack d e)', '(
    put-down d)', '(unstack e a)', '(stack e d)', '(pick-up a)', '(
    stack a b)']
Failed to validate the plan for problem blocks_instances/instance-455.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-458.
    pddl
['(pick-up e)', '(stack e b)', '(pick-up d)', '(stack d e)']
Failed to validate the plan for problem blocks_instances/instance-459.
    pddl
```

```
['(unstack e b)', '(unstack b a)', '(put-down b)', '(unstack a c)', '(
    put-down a)', '(pick-up b)', '(stack b a)', '(unstack c d)', '(
    pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-460.
    pddl
['(unstack a c)', '(put-down a)', '(unstack d e)', '(stack d a)', '(
    unstack e b)', '(put-down e)', '(pick-up b)', '(stack b c)', '(
    pick-up e)', '(stack e b)']
Failed to validate the plan for problem blocks_instances/instance-462.
    pddl
['(unstack e d)', '(put-down e)', '(unstack d a)', '(pick-up e)', '(
    stack e a)']
Failed to validate the plan for problem blocks_instances/instance-463.
    pddl
['(unstack d c)', '(put-down d)', '(unstack c b)', '(put-down c)', '(
    pick-up d)', '(stack d b)', '(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-465.
    pddl
['(pick-up c)', '(stack c e)', '(unstack b a)', '(stack b d)', '(pick-
    up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-466.
    pddl
['(pick-up a)', '(stack a c)', '(unstack e b)', '(put-down e)', '(
    unstack b d)', '(put-down b)', '(pick-up d)', '(stack d e)', '(
    pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-467.
    pddl
['(unstack d e)', '(put-down d)', '(unstack e b)', '(stack e a)', '(
    unstack b c)', '(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-468.
    pddl
['(unstack a e)', '(put-down a)', '(pick-up e)', '(stack e c)', '(
    unstack b d)', '(stack b e)', '(pick-up d)', '(stack d a)']
Failed to validate the plan for problem blocks_instances/instance-469.
    pddl
['(unstack e c)', '(unstack c a)', '(put-down c)', '(unstack a d)', '(
    stack a c)', '(pick-up d)', '(stack d a)', '(pick-up b)', '(stack
    b d)']
Failed to validate the plan for problem blocks_instances/instance-47.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a d)', '(stack a b)', '(
    pick-up d)', '(stack d a)', '(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-471.
    pddl
['(unstack d c)', '(stack d a)', '(pick-up e)', '(stack e c)']
Failed to validate the plan for problem blocks_instances/instance-472.
    pddl
['(unstack a c)', '(stack a b)', '(unstack c e)', '(put-down c)', '(
    unstack e d)', '(stack e c)', '(pick-up d)', '(stack d a)']
Failed to validate the plan for problem blocks_instances/instance-475.
    pddl
['(pick-up c)', '(stack c e)', '(pick-up d)', '(stack d c)', '(pick-up
     a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-478.
    pddl
['(unstack c d)', '(unstack d e)', '(put-down d)', '(unstack e a)', '(
    put-down e)', '(unstack a b)', '(stack a d)', '(pick-up e)', '(
    stack e a)', '(pick-up b)', '(stack b e)']
Failed to validate the plan for problem blocks_instances/instance-479.
    pddl
['(unstack a e)', '(put-down a)', '(unstack d c)', '(stack d a)', '(
    pick-up c)', '(stack c e)']
Failed to validate the plan for problem blocks_instances/instance-48.
    pddl
[]
```

```
Failed to validate the plan for problem blocks_instances/instance-480.
    pddl
['(unstack a e)', '(put-down a)', '(unstack e b)', '(stack e a)', '(
    pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-481.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a b)', '(stack a d)', '(
    unstack b c)', '(stack b e)', '(pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-482.
    pddl
['(unstack a d)', '(unstack e c)', '(put-down e)', '(pick-up d)', '(
    stack d e)', '(unstack c b)', '(pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-483.
    pddl
['(unstack b a)', '(unstack a c)', '(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-484.
    pddl
['(unstack a e)', '(put-down a)', '(unstack d c)', '(put-down d)', '(
    pick-up e)', '(stack e c)', '(pick-up d)', '(stack d e)', '(pick-
    up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-485.
    pddl
['(unstack b e)', '(put-down b)', '(unstack e c)', '(stack e a)', '(
    pick-up c)', '(stack c d)', '(pick-up b)', '(stack b e)']
Failed to validate the plan for problem blocks_instances/instance-486.
    pddl
['(unstack c b)', '(put-down c)', '(unstack d a)', '(stack d c)', '(
    unstack a e)', '(stack a d)', '(pick-up b)', '(stack b e)']
Failed to validate the plan for problem blocks_instances/instance-487.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-489.
    pddl
['(unstack a e)', '(put-down a)', '(unstack e c)', '(stack e b)', '(
    pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-490.
    pddl
['(pick-up a)', '(stack a b)', '(pick-up d)', '(stack d a)', '(unstack
     e c)', '(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-493.
    pddl
['(unstack b d)', '(stack b a)', '(unstack e c)', '(put-down e)', '(
    pick-up d)', '(stack d c)', '(pick-up e)', '(stack e d)']
Failed to validate the plan for problem blocks_instances/instance-494.
    pddl
['(pick-up a)', '(stack a e)', '(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-498.
    pddl
['(unstack c d)', '(put-down c)', '(pick-up a)', '(stack a d)', '(
    unstack e b)', '(stack e a)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-499.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a e)', '(pick-up d)', '(
    stack d e)']
Failed to validate the plan for problem blocks_instances/instance-50.
    pddl
['(unstack b c)', '(stack b a)', '(pick-up c)', '(stack c b)', '(pick-
    up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-500.
    pddl
['(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-501.
    pddl
['(pick-up b)', '(stack b d)']
```

```
Failed to validate the plan for problem blocks_instances/instance-51.
    pddl
['(unstack a c)', '(put-down a)', '(unstack d b)', '(put-down d)', '(
    pick-up b)', '(stack b a)', '(pick-up c)', '(stack c b)', '(pick-
    up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-52.
    pddl
['(unstack c a)', '(put-down c)', '(pick-up a)', '(stack a c)', '(
    unstack d b)', '(pick-up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-53.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a b)', '(put-down a)', '(
    unstack b c)', '(stack b d)', '(pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-54.
    pddl
['(unstack d c)', '(put-down d)', '(pick-up a)', '(stack a d)', '(pick
    -up c)', '(stack c a)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-55.
    pddl
['(unstack d b)', '(put-down d)', '(unstack b c)', '(put-down b)', '(
    unstack c a)', '(stack c d)', '(pick-up a)', '(stack a c)', '(pick
    -up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-57.
    pddl
['(unstack c b)', '(put-down c)', '(unstack b d)', '(put-down b)', '(
    pick-up c)', '(stack c b)', '(unstack d a)', '(put-down d)', '(
    pick-up a)', '(stack a c)', '(pick-up d)', '(stack d a)']
Failed to validate the plan for problem blocks_instances/instance-58.
    pddl
['(unstack a b)', '(put-down a)', '(unstack b d)', '(pick-up a)', '(
    stack a d)']
Failed to validate the plan for problem blocks_instances/instance-6.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a c)', '(put-down a)', '(
    unstack c b)', '(stack c d)', '(pick-up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-60.
    pddl
['(unstack d c)', '(unstack c a)', '(put-down c)', '(pick-up a)', '(
    stack a b)', '(pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-61.
    pddl
['(pick-up a)', '(stack a c)', '(unstack d b)', '(stack d a)']
Failed to validate the plan for problem blocks_instances/instance-62.
    pddl
['(unstack b c)', '(put-down b)', '(pick-up a)', '(stack a b)', '(
    unstack c d)', '(put-down c)', '(pick-up d)', '(stack d a)', '(
    pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-63.
    pddl
['(unstack a c)', '(pick-up c)', '(stack c b)', '(pick-up d)', '(stack
     d c)']
Failed to validate the plan for problem blocks_instances/instance-64.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-65.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-66.
    pddl
['(unstack a c)', '(put-down a)', '(unstack c d)', '(stack c b)', '(
    pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-68.
    pddl
['(unstack d a)', '(put-down d)', '(unstack a b)', '(put-down a)', '(
    unstack b c)', '(stack b d)', '(pick-up c)', '(stack c a)']
```

```
Failed to validate the plan for problem blocks_instances/instance-69.
    pddl
['(unstack a b)', '(put-down a)', '(unstack b d)', '(stack b c)', '(
    pick-up d)', '(stack d a)']
Failed to validate the plan for problem blocks_instances/instance-7.
    pddl
['(unstack a d)', '(put-down a)', '(unstack d b)', '(unstack b c)', '(
    pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-70.
    pddl
['(pick-up d)', '(stack d c)']
Failed to validate the plan for problem blocks_instances/instance-71.
    pddl
['(pick-up a)', '(stack a d)']
Failed to validate the plan for problem blocks_instances/instance-72.
    pddl
['(unstack a d)', '(pick-up d)', '(stack d b)']
Failed to validate the plan for problem blocks_instances/instance-73.
    pddl
['(unstack d c)', '(stack d a)', '(pick-up b)', '(stack b c)']
Failed to validate the plan for problem blocks_instances/instance-74.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-75.
    pddl
['(unstack d c)', '(put-down d)', '(unstack c a)', '(put-down c)', '(
    unstack a b)', '(stack a d)', '(pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-76.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-77.
    pddl
['(unstack a c)', '(put-down a)', '(unstack c b)', '(put-down c)', '(
    pick-up b)', '(stack b a)', '(pick-up d)', '(stack d b)', '(pick-
    up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-78.
    pddl
['(pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-8.
    pddl
['(unstack b c)', '(stack b a)', '(pick-up d)', '(stack d b)', '(pick-
    up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-82.
    pddl
['(unstack a c)', '(put-down a)', '(unstack c b)', '(stack c d)', '(
    pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-83.
    pddl
['(unstack b c)', '(put-down b)', '(unstack c a)', '(stack c d)', '(
    pick-up a)', '(stack a c)', '(pick-up b)', '(stack b a)']
Failed to validate the plan for problem blocks_instances/instance-85.
    pddl
['(unstack b a)', '(unstack a c)', '(put-down a)', '(pick-up c)', '(
    stack c d)', '(pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-86.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-87.
    pddl
['(unstack b a)', '(put-down b)', '(unstack a c)', '(put-down a)', '(
    unstack c d)', '(put-down c)', '(pick-up d)', '(stack d b)', '(
    pick-up a)', '(stack a d)', '(pick-up c)', '(stack c a)']
Failed to validate the plan for problem blocks_instances/instance-88.
    pddl
```

```
['(unstack d b)', '(put-down d)', '(unstack b c)', '(stack b a)', '(
    pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-89.
    pddl
['(unstack a d)', '(put-down a)', '(unstack b c)', '(stack b a)', '(
    pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-90.
    pddl
[]
Failed to validate the plan for problem blocks_instances/instance-91.
    pddl
['(unstack b a)', '(put-down b)', '(unstack a d)', '(put-down a)', '(
    unstack d c)', '(stack d a)', '(pick-up c)', '(stack c b)']
Failed to validate the plan for problem blocks_instances/instance-92.
    pddl
['(unstack d c)', '(put-down d)', '(unstack c b)', '(stack c a)', '(
    pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-93.
    pddl
['(unstack c a)', '(put-down c)', '(unstack a d)', '(put-down a)', '(
    pick-up c)', '(stack c d)', '(pick-up a)', '(stack a c)']
Failed to validate the plan for problem blocks_instances/instance-95.
    pddl
['(unstack b d)', '(put-down b)', '(unstack d a)', '(put-down d)', '(
    unstack a c)', '(stack a b)', '(pick-up c)', '(stack c d)']
Failed to validate the plan for problem blocks_instances/instance-96.
    pddl
['(unstack a d)', '(put-down a)', '(pick-up d)', '(stack d a)', '(
    unstack c b)', '(pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-97.
    pddl
['(unstack a c)', '(put-down a)', '(unstack c d)', '(stack c a)', '(
    pick-up b)', '(stack b d)']
Failed to validate the plan for problem blocks_instances/instance-99.
    pddl
['(unstack a d)', '(put-down a)', '(unstack d b)', '(stack d c)', '(
    pick-up b)', '(stack b a)']
Evaluating the final outcome
Success: 146, failure: 356, total expansions 95968, total generated
    283607
```

9.3 5x5 Crosswords

9.3.1 Deepseek-CoderV2

Listing 36: Without partial successor soundness test

```
AutoToS prompt:

The task is a 5x5 mini crosswords. A state is a 2D array representing
    the current puzzle state, where the initial grid is all "None".
    Note that some of the possible answers are not exactly 5 character
     long. Given an input of possible answers to horizontal clues and
    vertical clues, how is the successor state function defined?
    Please first think step by step. Then provide the successor state
    function in Python code.

The possible clues for each row and each column are given separately.
    Here is an example of possible horizontal and vertical clues:

horizontal_answers = [
        ["tasks", "goals", "plans", "agend", "chores", "works", "deeds
            ", "items", "lists", "brief"],
        ["motor", "power", "drive", "diesel", "steam", "pumps", "crank
            ", "gears", "turbn", "motor"],
```

```
        ["grand", "artsy", "showy", "ornate", "fancy", "vain", "proud", "vogue", "swank", "luxus"],
        ["venue", "salle", "forum", "atria", "lobby", "parls", "court", "malls", "mall", "lobby"],
        ["jeer", "scoff", "sleer", "deris", "sneer", "scorn", "derid", "gibes", "gibed", "flout"]
]
vertical_answers = [
        ["amass", "stack", "hoard", "pile", "store", "heaps", "massy", "gathe", "lumps", "mound"],
        ["nilga", "goral", "eland", "lepus", "gazal", "kudu", "oryx", "gnu", "imps", "carb"],
        ["scheme", "design", "ettle", "nettle", "sting", "wiles", "plans", "ideas", "plots", "cocks"],
        ["spout", "nosle", "snout", "mouth", "nostr", "ports", "inlet", "vents", "outlt", "beaks"],
        ["drier", "arid", "sere", "parch", "dryer", "wring", "drear", "sear", "pall", "lack"]
]
```

where horizontal_answers is a list where element i is a list of possible answers to clue in row i, and vertical_answers is a list where element i is a list of possible answers to clue in column i.

Model response:
To define the successor state function for a 5x5 mini crossword puzzle, we need to consider how to update the puzzle state based on the given possible answers for horizontal and vertical clues. The goal is to fill in the puzzle grid such that each row and column contains one of the provided possible answers.

Step-by-Step Thought Process:

1. **Initialization**: Start with an empty 5x5 grid where each cell is initialized to `None`.
2. **Selection of Answers**: For each row and each column, select one of the possible answers. The selection should be such that it does not conflict with the already placed answers in the grid.
3. **Placement of Answers**: Place the selected answer into the corresponding row or column of the grid. Ensure that the placement does not violate the constraints of the crossword (i.e., letters at intersections must match).
4. **Validation**: After placing an answer, validate the grid to ensure it remains consistent. If a conflict is detected, backtrack or try a different answer.
5. **Iteration**: Repeat the process until the grid is fully populated or until no valid placements can be made.

Successor State Function in Python:

The following Python function `get_successor_state` takes the current state of the puzzle, the list of horizontal and vertical possible answers, and returns the next valid state of the puzzle:

python
```
def get_successor_state(current_state, horizontal_answers, vertical_answers):
    def is_valid_placement(state, answer, row_or_col, is_row):
        if is_row:
            for i in range(len(answer)):
                if state[row_or_col][i] is not None and state[row_or_col][i] != answer[i]:
                    return False
        else:
```

```
            for i in range(len(answer)):
                if state[i][row_or_col] is not None and state[i][
                    row_or_col] != answer[i]:
                    return False
        return True

    def place_answer(state, answer, row_or_col, is_row):
        if is_row:
            for i in range(len(answer)):
                state[row_or_col][i] = answer[i]
        else:
            for i in range(len(answer)):
                state[i][row_or_col] = answer[i]

    def remove_answer(state, answer, row_or_col, is_row):
        if is_row:
            for i in range(len(answer)):
                state[row_or_col][i] = None
        else:
            for i in range(len(answer)):
                state[i][row_or_col] = None

    for row in range(5):
        for answer in horizontal_answers[row]:
            if is_valid_placement(current_state, answer, row, True):
                place_answer(current_state, answer, row, True)
                break

    for col in range(5):
        for answer in vertical_answers[col]:
            if is_valid_placement(current_state, answer, col, False):
                place_answer(current_state, answer, col, False)
                break

    return current_state

# Example usage:
initial_state = [[None for _ in range(5)] for _ in range(5)]
successor_state = get_successor_state(initial_state,
    horizontal_answers, vertical_answers)
print(successor_state)
```

Explanation:

- **is_valid_placement**: Checks if placing an answer in a specific row or column is valid without conflicts.
- **place_answer**: Places the answer in the specified row or column.
- **remove_answer**: Removes the answer from the specified row or column (useful for backtracking, which is not implemented here but can be added for a complete solution).
- **get_successor_state**: Iterates through each row and column, trying to place valid answers from the provided lists.

This function assumes that there is at least one valid answer for each row and column. If conflicts arise or no valid answer can be found, additional logic (like backtracking) would be needed to handle those cases

AutoToS prompt:

Please provide a goal test function in Python ensuring that all cells are not None, all words horizontally are matching at least one of the horizontal answers, and all words vertically match at least one of the vertical answers. The function should receive a state, horizontal_answers, and vertical_answers as before and return True for a goal state and False otherwise.

Model response:
Certainly! The goal test function will check if the current state of the puzzle meets the criteria for being a goal state. This includes ensuring that all cells are filled (not ‘None‘), and that each row and column contains a word that matches at least one of the provided possible answers for horizontal and vertical clues, respectively.

Here’s the Python function ‘is_goal_state‘ that performs this check:

python
```
def is_goal_state(state, horizontal_answers, vertical_answers):
    # Check if all cells are filled
    for row in state:
        if None in row:
            return False

    # Check if each row matches at least one of the horizontal answers
    for i, row in enumerate(state):
        row_word = ''.join(row)
        if row_word not in horizontal_answers[i]:
            return False

    # Check if each column matches at least one of the vertical
       answers
    for j in range(len(state[0])):
        col_word = ''.join(state[i][j] for i in range(len(state)))
        if col_word not in vertical_answers[j]:
            return False

    # If all checks pass, the state is a goal state
    return True

# Example usage:
goal_state = [
    ["tasks", "motor", "grand", "venue", "jeer"],
    ["amass", "nilga", "scheme", "spout", "drier"],
    # Fill in the rest of the rows to match the example provided
]

print(is_goal_state(goal_state, horizontal_answers, vertical_answers))
```

Explanation:

1. **Check if all cells are filled**: The function first iterates through each row of the state. If any cell contains ‘None‘, it returns ‘False‘ immediately, indicating that the state is not a goal state.

2. **Check horizontal matches**: For each row, the function constructs a string (‘row_word‘) from the cells of that row. It then checks if this string is present in the list of possible horizontal answers for that row. If not, it returns ‘False‘.

3. **Check vertical matches**: For each column, the function constructs a string (‘col_word‘) from the cells of that column. It then checks if this string is present in the list of possible vertical answers for that column. If not, it returns ‘False‘.

4. **Return True if all checks pass**: If all rows and columns pass the checks, the function returns 'True', indicating that the state is a goal state.

This function assumes that the state is a valid 5x5 grid and that the 'horizontal_answers' and 'vertical_answers' lists are correctly formatted as described

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
Input state should not change as a result of applying the successor function.
State before successor function applied: [[None, None, None, None, None], [None, None, None, None, None], [None, None, None, None, None], [None, None, None, None, None], [None, None, None, None, None]]
State after successor function applied: [['t', 'a', 's', 'k', 's'], ['m', 'o', 't', 'o', 'r'], ['g', 'r', 'a', 'n', 'd'], ['v', 'e', 'n', 'u', 'e'], ['j', 'e', 'e', 'r', None]]
Let's think step by step. First, think of the possible reasons why the input state may change as a result of applying the successor function, such as the use of shallow copy.
Then, provide the complete Python code for the revised successor function that returns a list of successor states and ensure it does not change the input state.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.
Horizontal clues: [['tasks', 'goals', 'plans', 'agend', 'chores', 'works', 'deeds', 'items', 'lists', 'brief'], ['motor', 'power', 'drive', 'diesel', 'steam', 'pumps', 'crank', 'gears', 'turbn', 'motor'], ['grand', 'artsy', 'showy', 'ornate', 'fancy', 'vain', 'proud', 'vogue', 'swank', 'luxus'], ['venue', 'salle', 'forum', 'atria', 'lobby', 'parls', 'court', 'malls', 'mall', 'lobby'], ['jeer', 'scoff', 'sleer', 'deris', 'sneer', 'scorn', 'derid', 'gibes', 'gibed', 'flout']]
Vertical clues: [['amass', 'stack', 'hoard', 'pile', 'store', 'heaps', 'massy', 'gathe', 'lumps', 'mound'], ['nilga', 'goral', 'eland', 'lepus', 'gazal', 'kudu', 'oryx', 'gnu', 'imps', 'carb'], ['scheme', 'design', 'ettle', 'nettle', 'sting', 'wiles', 'plans', 'ideas', 'plots', 'cocks'], ['spout', 'nosle', 'snout', 'mouth', 'nostr', 'ports', 'inlet', 'vents', 'outlt', 'beaks'], ['drier', 'arid', 'sere', 'parch', 'dryer', 'wring', 'drear', 'sear', 'pall', 'lack']]

Model response:
To ensure that the input state does not change as a result of applying the successor function, we need to make sure that we are working with a deep copy of the state rather than a shallow copy. A shallow copy would only copy the references to the lists within the state, meaning changes to the inner lists would affect the original state. A deep copy, on the other hand, creates a completely independent copy of the state, including all nested structures.

Here's the revised successor function that returns a list of successor states without modifying the input state:

python
```
import copy
```

```
def get_successor_state(current_state, horizontal_answers,
    vertical_answers):
    def is_valid_placement(state, answer, row_or_col, is_row):
        if is_row:
            for i in range(len(answer)):
                if state[row_or_col][i] is not None and state[
                    row_or_col][i] != answer[i]:
                    return False
        else:
            for i in range(len(answer)):
                if state[i][row_or_col] is not None and state[i][
                    row_or_col] != answer[i]:
                    return False
        return True

    def place_answer(state, answer, row_or_col, is_row):
        new_state = copy.deepcopy(state)
        if is_row:
            for i in range(len(answer)):
                new_state[row_or_col][i] = answer[i]
        else:
            for i in range(len(answer)):
                new_state[i][row_or_col] = answer[i]
        return new_state

    successor_states = []

    for row in range(5):
        for answer in horizontal_answers[row]:
            if is_valid_placement(current_state, answer, row, True):
                new_state = place_answer(current_state, answer, row,
                    True)
                successor_states.append(new_state)

    for col in range(5):
        for answer in vertical_answers[col]:
            if is_valid_placement(current_state, answer, col, False):
                new_state = place_answer(current_state, answer, col,
                    False)
                successor_states.append(new_state)

    return successor_states

# Example usage:
initial_state = [[None for _ in range(5)] for _ in range(5)]
successor_states = get_successor_state(initial_state,
    horizontal_answers, vertical_answers)
for state in successor_states:
    print(state)
```

Explanation:

1. **Deep Copy**: The `place_answer` function now creates a deep copy of the state using `copy.deepcopy` before placing the answer. This ensures that the original state remains unchanged.

2. **Return List of States**: The function now returns a list of successor states, each representing a possible next state of the puzzle.

3. **Validation and Placement**: The `is_valid_placement` function checks if placing an answer is valid, and `place_answer` places the answer in the deep-copied state.

This revised function ensures that the input state is not modified and returns a list of valid successor states based on the provided horizontal and vertical clues
Successor Iteration 2

AutoToS prompt:
The following exception was produced when testing the successor function:
  File "<string>", line 9, in is_valid_placement

list index out of range. Please fix this exception in the successor function.
Model response:
The 'list index out of range' exception occurs when the function tries to access an index that is beyond the length of the list. This can happen if the length of the answer (word) being placed is less than 5 characters, causing an out-of-bounds access when trying to place it in the 5x5 grid.

To fix this, we need to ensure that the length of the answer being placed matches the length of the row or column it is being placed in. If the answer is shorter than 5 characters, it should be padded with 'None' or another placeholder to match the length of the grid.

Here's the revised successor function with the necessary checks and fixes:

python
```
import copy

def get_successor_state(current_state, horizontal_answers,
    vertical_answers):
    def is_valid_placement(state, answer, row_or_col, is_row):
        if is_row:
            if len(answer) != len(state[row_or_col]):
                return False
            for i in range(len(answer)):
                if state[row_or_col][i] is not None and state[
                    row_or_col][i] != answer[i]:
                    return False
        else:
            if len(answer) != len(state):
                return False
            for i in range(len(answer)):
                if state[i][row_or_col] is not None and state[i][
                    row_or_col] != answer[i]:
                    return False
        return True

    def place_answer(state, answer, row_or_col, is_row):
        new_state = copy.deepcopy(state)
        if is_row:
            for i in range(len(answer)):
                new_state[row_or_col][i] = answer[i]
        else:
            for i in range(len(answer)):
                new_state[i][row_or_col] = answer[i]
        return new_state

    successor_states = []

    for row in range(5):
        for answer in horizontal_answers[row]:
```

```
            if is_valid_placement(current_state, answer, row, True):
                new_state = place_answer(current_state, answer, row,
                    True)
                successor_states.append(new_state)

    for col in range(5):
        for answer in vertical_answers[col]:
            if is_valid_placement(current_state, answer, col, False):
                new_state = place_answer(current_state, answer, col,
                    False)
                successor_states.append(new_state)

    return successor_states

# Example usage:
initial_state = [[None for _ in range(5)] for _ in range(5)]
successor_states = get_successor_state(initial_state,
    horizontal_answers, vertical_answers)
for state in successor_states:
    print(state)
```

```
### Explanation:

1. **Length Check**: The `is_valid_placement` function now includes
    checks to ensure that the length of the answer matches the length
    of the row or column it is being placed in. If the lengths do not
    match, the placement is considered invalid.

2. **Deep Copy**: The `place_answer` function continues to use `copy.
    deepcopy` to create a deep copy of the state before placing the
    answer, ensuring that the original state remains unchanged.

3. **Return List of States**: The function returns a list of successor
     states, each representing a possible next state of the puzzle.

This revised function should no longer produce the `list index out of
    range` exception and will correctly handle answers of varying
    lengths by ensuring they match the dimensions of the 5x5 grid

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 20, failure: 0, total expansions 462616, total generated
    3904718
```

Listing 37: With partial successor soundness test

```
AutoToS prompt:

The task is a 5x5 mini crosswords. A state is a 2D array representing
    the current puzzle state, where the initial grid is all "None".
    Note that some of the possible answers are not exactly 5 character
     long. Given an input of possible answers to horizontal clues and
    vertical clues, how is the successor state function defined?
    Please first think step by step. Then provide the successor state
    function in Python code.

The possible clues for each row and each column are given separately.
    Here is an example of possible horizontal and vertical clues:

horizontal_answers = [
```

```
        ["tasks", "goals", "plans", "agend", "chores", "works", "deeds
            ", "items", "lists", "brief"],
        ["motor", "power", "drive", "diesel", "steam", "pumps", "crank
            ", "gears", "turbn", "motor"],
        ["grand", "artsy", "showy", "ornate", "fancy", "vain", "proud
            ", "vogue", "swank", "luxus"],
        ["venue", "salle", "forum", "atria", "lobby", "parls", "court
            ", "malls", "mall", "lobby"],
        ["jeer", "scoff", "sleer", "deris", "sneer", "scorn", "derid",
             "gibes", "gibed", "flout"]
]
vertical_answers = [
        ["amass", "stack", "hoard", "pile", "store", "heaps", "massy",
             "gathe", "lumps", "mound"],
        ["nilga", "goral", "eland", "lepus", "gazal", "kudu", "oryx",
            "gnu", "imps", "carb"],
        ["scheme", "design", "ettle", "nettle", "sting", "wiles", "
            plans", "ideas", "plots", "cocks"],
        ["spout", "nosle", "snout", "mouth", "nostr", "ports", "inlet
            ", "vents", "outlt", "beaks"],
        ["drier", "arid", "sere", "parch", "dryer", "wring", "drear",
            "sear", "pall", "lack"]
]
```

where horizontal_answers is a list where element i is a list of possible answers to clue in row i, and vertical_answers is a list where element i is a list of possible answers to clue in column i.

Model response:

To define the successor state function for a 5x5 mini crossword puzzle, we need to consider how to update the puzzle state based on the given possible answers for horizontal and vertical clues. The goal is to fill in the puzzle grid one cell at a time, ensuring that each cell is filled correctly according to the constraints provided by the possible answers.

Step-by-Step Approach:

1. **Initialization**: Start with an empty 5x5 grid (all cells set to 'None').
2. **Selection**: Choose a row or column to fill next. This choice can be made based on the current state of the puzzle, aiming to minimize conflicts or maximize progress.
3. **Validation**: For the chosen row or column, check each possible answer to see if it fits within the current state of the puzzle without conflicts.
4. **Update**: If a valid answer is found, update the corresponding row or column in the puzzle grid.
5. **Iteration**: Repeat the process until the puzzle is fully solved or no further progress can be made.

Successor State Function in Python:

The following Python function 'get_successor_state' implements the successor state logic:

python
```
def get_successor_state(current_state, horizontal_answers,
    vertical_answers):
    def is_valid_placement(state, answer, row_or_col, is_row):
        if is_row:
            for i in range(len(answer)):
                if state[row_or_col][i] is not None and state[
                    row_or_col][i] != answer[i]:
```

```
                return False
        else:
            for i in range(len(answer)):
                if state[i][row_or_col] is not None and state[i][
                    row_or_col] != answer[i]:
                    return False
        return True

    def place_answer(state, answer, row_or_col, is_row):
        if is_row:
            for i in range(len(answer)):
                state[row_or_col][i] = answer[i]
        else:
            for i in range(len(answer)):
                state[i][row_or_col] = answer[i]

    for row in range(5):
        for answer in horizontal_answers[row]:
            if len(answer) <= 5 and is_valid_placement(current_state,
                answer, row, True):
                new_state = [row[:] for row in current_state]
                place_answer(new_state, answer, row, True)
                yield new_state

    for col in range(5):
        for answer in vertical_answers[col]:
            if len(answer) <= 5 and is_valid_placement(current_state,
                answer, col, False):
                new_state = [row[:] for row in current_state]
                place_answer(new_state, answer, col, False)
                yield new_state

# Example usage:
initial_state = [[None] * 5 for _ in range(5)]
for state in get_successor_state(initial_state, horizontal_answers,
    vertical_answers):
    for row in state:
        print(row)
    print("-----")
```

Explanation:

1. **is_valid_placement**: This helper function checks if a given answer can be placed in a specified row or column without conflicts.
2. **place_answer**: This helper function places a valid answer into the specified row or column of the puzzle grid.
3. **get_successor_state**: This function iterates over each row and column, trying to place each possible answer. If a valid placement is found, it yields the updated state.

This approach ensures that the puzzle is filled incrementally, respecting the constraints provided by the possible answers for each row and column

AutoToS prompt:

Please provide a goal test function in Python ensuring that all cells are not None, all words horizontally are matching at least one of the horizontal answers, and all words vertically match at least one of the vertical answers. The function should receive a state, horizontal_answers, and vertical_answers as before and return True for a goal state and False otherwise.

Model response:
Certainly! Below is the Python function ‘is_goal_state‘ that checks if a given state of the 5x5 mini crossword puzzle meets the criteria for being a goal state:

python
```
def is_goal_state(state, horizontal_answers, vertical_answers):
    # Check if all cells are not None
    for row in state:
        if None in row:
            return False

    # Check if all horizontal words match at least one of the horizontal answers
    for i, row in enumerate(state):
        word = ’’.join(row)
        if word not in horizontal_answers[i]:
            return False

    # Check if all vertical words match at least one of the vertical answers
    for j in range(5):
        word = ’’.join(state[i][j] for i in range(5))
        if word not in vertical_answers[j]:
            return False

    return True

# Example usage:
goal_state = [
    [’t’, ’a’, ’s’, ’k’, ’s’],
    [’m’, ’o’, ’t’, ’o’, ’r’],
    [’g’, ’r’, ’a’, ’n’, ’d’],
    [’v’, ’e’, ’n’, ’u’, ’e’],
    [’j’, ’e’, ’e’, ’r’, ’s’]
]

print(is_goal_state(goal_state, horizontal_answers, vertical_answers))  # Should return True
```

Explanation:

1. **Check for None Cells**: The function first iterates through each row of the state to ensure that no cell contains ‘None‘. If any cell is ‘None‘, it returns ‘False‘.

2. **Check Horizontal Words**: For each row, the function constructs the word by joining the characters in the row and checks if this word is present in the corresponding list of horizontal answers. If any word does not match, it returns ‘False‘.

3. **Check Vertical Words**: For each column, the function constructs the word by joining the characters in the column and checks if this word is present in the corresponding list of vertical answers . If any word does not match, it returns ‘False‘.

4. **Return True**: If all checks pass, the function returns ‘True‘, indicating that the state is a goal state.

This function ensures that the puzzle is fully solved and that all words match the given clues for both horizontal and vertical directions

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
Successor state has the same number of filled cells as the parent state.
Let’s think step by step. First, think what you did wrong.
Then, think of in what ways successor state should be different from the parent state.
Then, provide the complete Python code for the revised successor function that returns a list of successor states.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: [[None, None, None, None, ’l’], [None, None, None, None, ’a’], [None, None, None, None, ’c’], [None, None, None, None, ’k’], [None, None, None, None, None]]
Example wrong successor state: [[None, None, None, None, ’l’], [None, None, None, None, ’a’], [None, None, None, None, ’c’], [None, None, None, None, ’k’], [None, None, None, None, None]]
Horizontal clues: [[’tasks’, ’goals’, ’plans’, ’agend’, ’chores’, ’works’, ’deeds’, ’items’, ’lists’, ’brief’], [’motor’, ’power’, ’drive’, ’diesel’, ’steam’, ’pumps’, ’crank’, ’gears’, ’turbn’, ’motor’], [’grand’, ’artsy’, ’showy’, ’ornate’, ’fancy’, ’vain’, ’proud’, ’vogue’, ’swank’, ’luxus’], [’venue’, ’salle’, ’forum’, ’atria’, ’lobby’, ’parls’, ’court’, ’malls’, ’mall’, ’lobby’], [’jeer’, ’scoff’, ’sleer’, ’deris’, ’sneer’, ’scorn’, ’derid’, ’gibes’, ’gibed’, ’flout’]]
Vertical clues: [[’amass’, ’stack’, ’hoard’, ’pile’, ’store’, ’heaps’, ’massy’, ’gathe’, ’lumps’, ’mound’], [’nilga’, ’goral’, ’eland’, ’lepus’, ’gazal’, ’kudu’, ’oryx’, ’gnu’, ’imps’, ’carb’], [’scheme’, ’design’, ’ettle’, ’nettle’, ’sting’, ’wiles’, ’plans’, ’ideas’, ’plots’, ’cocks’], [’spout’, ’nosle’, ’snout’, ’mouth’, ’nostr’, ’ports’, ’inlet’, ’vents’, ’outlt’, ’beaks’], [’drier’, ’arid’, ’sere’, ’parch’, ’dryer’, ’wring’, ’drear’, ’sear’, ’pall’, ’lack’]]

Model response:
Step-by-Step Analysis and Revision

What Went Wrong:
The initial successor state function did not ensure that each successor state had a different number of filled cells compared to the parent state. This means that the function was not progressing the state towards a solution but rather generating states that were identical to the parent state.

How to Make Successor State Different:
To ensure that each successor state is different from the parent state, we need to:
1. Only attempt to place words in rows or columns that have empty cells.
2. Ensure that placing a word in a row or column changes the state by filling in at least one ‘None‘ cell.
3. Avoid generating states that are identical to the parent state.

Revised Successor State Function:
The revised function will focus on filling in the next available ‘None’ cell by trying to place valid words from the horizontal and vertical clues.

python
```
```

```
def get_successor_state(current_state, horizontal_answers,
    vertical_answers):
    def is_valid_placement(state, answer, row_or_col, is_row):
        if is_row:
            for i in range(len(answer)):
                if state[row_or_col][i] is not None and state[
                    row_or_col][i] != answer[i]:
                    return False
        else:
            for i in range(len(answer)):
                if state[i][row_or_col] is not None and state[i][
                    row_or_col] != answer[i]:
                    return False
        return True

    def place_answer(state, answer, row_or_col, is_row):
        if is_row:
            for i in range(len(answer)):
                state[row_or_col][i] = answer[i]
        else:
            for i in range(len(answer)):
                state[i][row_or_col] = answer[i]

    successor_states = []

    # Try to place horizontal answers
    for row in range(5):
        if None in current_state[row]:  # Only attempt to place if
            there are None cells
            for answer in horizontal_answers[row]:
                if len(answer) <= 5 and is_valid_placement(
                    current_state, answer, row, True):
                    new_state = [row[:] for row in current_state]
                    place_answer(new_state, answer, row, True)
                    if new_state != current_state:  # Ensure the state
                         has changed
                        successor_states.append(new_state)

    # Try to place vertical answers
    for col in range(5):
        if any(current_state[row][col] is None for row in range(5)):
            # Only attempt to place if there are None cells
            for answer in vertical_answers[col]:
                if len(answer) <= 5 and is_valid_placement(
                    current_state, answer, col, False):
                    new_state = [row[:] for row in current_state]
                    place_answer(new_state, answer, col, False)
                    if new_state != current_state:  # Ensure the state
                         has changed
                        successor_states.append(new_state)

    return successor_states

# Example usage:
initial_state = [
    [None, None, None, None, 'l'],
    [None, None, None, None, 'a'],
    [None, None, None, None, 'c'],
    [None, None, None, None, 'k'],
    [None, None, None, None, None]
]

successor_states = get_successor_state(initial_state,
    horizontal_answers, vertical_answers)
for state in successor_states:
```

```
    for row in state:
        print(row)
    print("-----")
```

```
### Explanation:
1. **is_valid_placement**: Checks if an answer can be placed in a
   specified row or column without conflicts.
2. **place_answer**: Places a valid answer into the specified row or
   column of the puzzle grid.
3. **get_successor_state**: Iterates over each row and column, trying
   to place each possible answer. It ensures that only rows or
   columns with 'None' cells are attempted, and it only adds states
   to the successor list if they are different from the current state
   .

This revised function ensures that each successor state has a
   different number of filled cells compared to the parent state,
   progressing the puzzle towards a solution

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 20, failure: 0, total expansions 1587973, total generated
   5939765
```

9.3.2 GPT4-Mini

Listing 38: Without partial successor soundness test

```
AutoToS prompt:

The task is a 5x5 mini crosswords. A state is a 2D array representing
   the current puzzle state, where the initial grid is all "None".
   Note that some of the possible answers are not exactly 5 character
    long. Given an input of possible answers to horizontal clues and
   vertical clues, how is the successor state function defined?
   Please first think step by step. Then provide the successor state
   function in Python code.

The possible clues for each row and each column are given separately.
   Here is an example of possible horizontal and vertical clues:

horizontal_answers = [
        ["tasks", "goals", "plans", "agend", "chores", "works", "deeds
           ", "items", "lists", "brief"],
        ["motor", "power", "drive", "diesel", "steam", "pumps", "crank
           ", "gears", "turbn", "motor"],
        ["grand", "artsy", "showy", "ornate", "fancy", "vain", "proud
           ", "vogue", "swank", "luxus"],
        ["venue", "salle", "forum", "atria", "lobby", "parls", "court
           ", "malls", "mall", "lobby"],
        ["jeer", "scoff", "sleer", "deris", "sneer", "scorn", "derid",
            "gibes", "gibed", "flout"]
]
vertical_answers = [
        ["amass", "stack", "hoard", "pile", "store", "heaps", "massy",
            "gathe", "lumps", "mound"],
        ["nilga", "goral", "eland", "lepus", "gazal", "kudu", "oryx",
           "gnu", "imps", "carb"],
        ["scheme", "design", "ettle", "nettle", "sting", "wiles", "
           plans", "ideas", "plots", "cocks"],
```

```
        ["spout", "nosle", "snout", "mouth", "nostr", "ports", "inlet
            ", "vents", "outlt", "beaks"],
        ["drier", "arid", "sere", "parch", "dryer", "wring", "drear",
            "sear", "pall", "lack"]
]
```

where horizontal_answers is a list where element i is a list of possible answers to clue in row i, and vertical_answers is a list where element i is a list of possible answers to clue in column i.

Model response:

```python
def successor_state(state, horizontal_answers, vertical_answers):
    from typing import List, Optional

    def is_valid_placement(state: List[List[Optional[str]]], word: str
        , row: int, col: int, is_horizontal: bool) -> bool:
        length = len(word)
        if is_horizontal:
            if col + length > len(state[0]) or any(state[row][col + i]
                 not in {None, word[i]} for i in range(length)):
                return False
        else:
            if row + length > len(state) or any(state[row + i][col]
                not in {None, word[i]} for i in range(length)):
                return False
        return True

    def place_word(state: List[List[Optional[str]]], word: str, row:
        int, col: int, is_horizontal: bool) -> List[List[Optional[str
        ]]]:
        new_state = [row[:] for row in state]
        length = len(word)
        if is_horizontal:
            for i in range(length):
                new_state[row][col + i] = word[i]
        else:
            for i in range(length):
                new_state[row + i][col] = word[i]
        return new_state

    new_states = []

    # Try placing each horizontal answer
    for row in range(len(horizontal_answers)):
        for answer in horizontal_answers[row]:
            if is_valid_placement(state, answer, row, 0, True):
                new_states.append(place_word(state, answer, row, 0,
                    True))

    # Try placing each vertical answer
    for col in range(len(vertical_answers)):
        for answer in vertical_answers[col]:
            if is_valid_placement(state, answer, 0, col, False):
                new_states.append(place_word(state, answer, 0, col,
                    False))

    return new_states
```

AutoToS prompt:

Please provide a goal test function in Python ensuring that all cells are not None, all words horizontally are matching at least one of the horizontal answers, and all words vertically match at least

```
    one of the vertical answers. The function should receive a state,
    horizontal_answers, and vertical_answers as before and return True
     for a goal state and False otherwise.

Model response:
```python
def goal_test(state, horizontal_answers, vertical_answers):
    from typing import List, Optional

    def matches_horizontal(state: List[List[Optional[str]]], row: int)
         -> bool:
        word = ''.join(state[row])
        return any(word == answer for answer in horizontal_answers[row
            ])

    def matches_vertical(state: List[List[Optional[str]]], col: int)
        -> bool:
        word = ''.join(state[row][col] for row in range(len(state)))
        return any(word == answer for answer in vertical_answers[col])

    # Check if all cells are filled
    for row in state:
        if any(cell is None for cell in row):
            return False

    # Check horizontally
    for row in range(len(state)):
        if not matches_horizontal(state, row):
            return False

    # Check vertically
    for col in range(len(state[0])):
        if not matches_vertical(state, col):
            return False

    return True
```

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 20, failure: 0, total expansions 1587973, total generated
    13705798
```

Listing 39: With partial successor soundness test

```
AutoToS prompt:

The task is a 5x5 mini crosswords. A state is a 2D array representing
    the current puzzle state, where the initial grid is all "None".
    Note that some of the possible answers are not exactly 5 character
     long. Given an input of possible answers to horizontal clues and
    vertical clues, how is the successor state function defined?
    Please first think step by step. Then provide the successor state
    function in Python code.

The possible clues for each row and each column are given separately.
    Here is an example of possible horizontal and vertical clues:
```

```
horizontal_answers = [
        ["tasks", "goals", "plans", "agend", "chores", "works", "deeds
            ", "items", "lists", "brief"],
        ["motor", "power", "drive", "diesel", "steam", "pumps", "crank
            ", "gears", "turbn", "motor"],
        ["grand", "artsy", "showy", "ornate", "fancy", "vain", "proud
            ", "vogue", "swank", "luxus"],
        ["venue", "salle", "forum", "atria", "lobby", "parls", "court
            ", "malls", "mall", "lobby"],
        ["jeer", "scoff", "sleer", "deris", "sneer", "scorn", "derid",
             "gibes", "gibed", "flout"]
]
vertical_answers = [
        ["amass", "stack", "hoard", "pile", "store", "heaps", "massy",
             "gathe", "lumps", "mound"],
        ["nilga", "goral", "eland", "lepus", "gazal", "kudu", "oryx",
            "gnu", "imps", "carb"],
        ["scheme", "design", "ettle", "nettle", "sting", "wiles", "
            plans", "ideas", "plots", "cocks"],
        ["spout", "nosle", "snout", "mouth", "nostr", "ports", "inlet
            ", "vents", "outlt", "beaks"],
        ["drier", "arid", "sere", "parch", "dryer", "wring", "drear",
            "sear", "pall", "lack"]
]
```

where horizontal_answers is a list where element i is a list of possible answers to clue in row i, and vertical_answers is a list where element i is a list of possible answers to clue in column i.

Model response:

```
```python
def successor_state(state, horizontal_answers, vertical_answers):
    from itertools import product

    def is_valid(state, row, col, word, horizontal=True):
        if horizontal:
            for idx in range(len(word)):
                if state[row][col + idx] is not None and state[row][
                    col + idx] != word[idx]:
                    return False
            return True
        else:
            for idx in range(len(word)):
                if state[row + idx][col] is not None and state[row +
                    idx][col] != word[idx]:
                    return False
            return True

    def place_word(state, row, col, word, horizontal=True):
        new_state = [row[:] for row in state]
        if horizontal:
            for idx in range(len(word)):
                new_state[row][col + idx] = word[idx]
        else:
            for idx in range(len(word)):
                new_state[row + idx][col] = word[idx]
        return new_state

    successors = []
    for row, col in product(range(5), range(5)):
        if state[row][col] is None:  # Only try to place a word if the
             cell is empty
            for word in horizontal_answers[row]:
                if is_valid(state, row, col, word, horizontal=True):
```
```

```
                    new_state = place_word(state, row, col, word,
                        horizontal=True)
                    successors.append(new_state)
            for word in vertical_answers[col]:
                if is_valid(state, row, col, word, horizontal=False):
                    new_state = place_word(state, row, col, word,
                        horizontal=False)
                    successors.append(new_state)

    return successors
```

AutoToS prompt:

Please provide a goal test function in Python ensuring that all cells are not None, all words horizontally are matching at least one of the horizontal answers, and all words vertically match at least one of the vertical answers. The function should receive a state, horizontal_answers, and vertical_answers as before and return True for a goal state and False otherwise.

Model response:
```python
def is_goal_state(state, horizontal_answers, vertical_answers):
    def word_in_list(word, word_list):
        return word in word_list

    def check_horizontal(state, row):
        word = ''.join(state[row])
        return any(word_in_list(word, answers) for answers in
            horizontal_answers[row])

    def check_vertical(state, col):
        word = ''.join(state[row][col] for row in range(5))
        return any(word_in_list(word, answers) for answers in
            vertical_answers[col])

    # Check if all cells are not None
    for row in state:
        if any(cell is None for cell in row):
            return False

    # Check horizontal words
    for row in range(5):
        if not check_horizontal(state, row):
            return False

    # Check vertical words
    for col in range(5):
        if not check_vertical(state, col):
            return False

    return True
```

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
The following exception was produced when testing the successor function:
  File "<string>", line 7, in is_valid

list index out of range. Please fix this exception in the successor function.
Model response:

```python
def successor_state(state, horizontal_answers, vertical_answers):
    from itertools import product

    def is_valid(state, row, col, word, horizontal=True):
        if horizontal:
            if col + len(word) > 5:  # Check if the word can fit horizontally
                return False
            for idx in range(len(word)):
                if state[row][col + idx] is not None and state[row][col + idx] != word[idx]:
                    return False
            return True
        else:
            if row + len(word) > 5:  # Check if the word can fit vertically
                return False
            for idx in range(len(word)):
                if state[row + idx][col] is not None and state[row + idx][col] != word[idx]:
                    return False
            return True

    def place_word(state, row, col, word, horizontal=True):
        new_state = [row[:] for row in state]
        if horizontal:
            for idx in range(len(word)):
                new_state[row][col + idx] = word[idx]
        else:
            for idx in range(len(word)):
                new_state[row + idx][col] = word[idx]
        return new_state

    successors = []
    for row, col in product(range(5), range(5)):
        if state[row][col] is None:  # Only try to place a word if the cell is empty
            for word in horizontal_answers[row]:  # Try to place horizontal words
                if is_valid(state, row, col, word, horizontal=True):
                    new_state = place_word(state, row, col, word, horizontal=True)
                    successors.append(new_state)
            for word in vertical_answers[col]:  # Try to place vertical words
                if is_valid(state, row, col, word, horizontal=False):
                    new_state = place_word(state, row, col, word, horizontal=False)
                    successors.append(new_state)

    return successors
```

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 20, failure: 0, total expansions 4514757, total generated 16830232

9.3.3 GPT4

Listing 40: Without partial successor soundness test

```
AutoToS prompt:

The task is a 5x5 mini crosswords. A state is a 2D array representing
    the current puzzle state, where the initial grid is all "None".
    Note that some of the possible answers are not exactly 5 character
     long. Given an input of possible answers to horizontal clues and
    vertical clues, how is the successor state function defined?
    Please first think step by step. Then provide the successor state
    function in Python code.

The possible clues for each row and each column are given separately.
    Here is an example of possible horizontal and vertical clues:

horizontal_answers = [
        ["tasks", "goals", "plans", "agend", "chores", "works", "deeds
            ", "items", "lists", "brief"],
        ["motor", "power", "drive", "diesel", "steam", "pumps", "crank
            ", "gears", "turbn", "motor"],
        ["grand", "artsy", "showy", "ornate", "fancy", "vain", "proud
            ", "vogue", "swank", "luxus"],
        ["venue", "salle", "forum", "atria", "lobby", "parls", "court
            ", "malls", "mall", "lobby"],
        ["jeer", "scoff", "sleer", "deris", "sneer", "scorn", "derid",
             "gibes", "gibed", "flout"]
]
vertical_answers = [
        ["amass", "stack", "hoard", "pile", "store", "heaps", "massy",
             "gathe", "lumps", "mound"],
        ["nilga", "goral", "eland", "lepus", "gazal", "kudu", "oryx",
            "gnu", "imps", "carb"],
        ["scheme", "design", "ettle", "nettle", "sting", "wiles", "
            plans", "ideas", "plots", "cocks"],
        ["spout", "nosle", "snout", "mouth", "nostr", "ports", "inlet
            ", "vents", "outlt", "beaks"],
        ["drier", "arid", "sere", "parch", "dryer", "wring", "drear",
            "sear", "pall", "lack"]
]

where horizontal_answers is a list where element i is a list of
    possible answers to clue in row i, and vertical_answers is a list
    where element i is a list of possible answers to clue in column i.

Model response:
To solve this problem step by step, we need to develop a function to
    generate a potential successor state of the crossword puzzle given
     the current state and possible horizontal and vertical answers.
    The function will fill the grid incrementally and generate the
    next state.

Here’s the approach we will take:
1. Iterate through each cell in the current crossword grid.
2. For each cell, identify the row and column it belongs to.
3. Check the possible horizontal answers for the row and vertical
    answers for the column.
4. Generate potential next states by filling in words that comply with
     the given constraints.
5. Return a list of all valid next states.

Let’s include the necessary imports and define the function that
    addresses these steps:
```

```python
from copy import deepcopy

def get_successor_states(grid, horizontal_answers, vertical_answers):
    def validate_word(grid, word, r, c, direction):
        if direction == 'horizontal':
            if c + len(word) > len(grid[0]):
                return False
            for i, char in enumerate(word):
                if grid[r][c + i] is not None and grid[r][c + i] != char:
                    return False
        elif direction == 'vertical':
            if r + len(word) > len(grid):
                return False
            for i, char in enumerate(word):
                if grid[r + i][c] is not None and grid[r + i][c] != char:
                    return False
        return True

    def fill_word(grid, word, r, c, direction):
        new_grid = deepcopy(grid)
        if direction == 'horizontal':
            for i, char in enumerate(word):
                new_grid[r][c + i] = char
        elif direction == 'vertical':
            for i, char in enumerate(word):
                new_grid[r + i][c] = char
        return new_grid

    n_rows, n_cols = len(grid), len(grid[0])
    successor_states = []

    for r in range(n_rows):
        for c in range(n_cols):
            if grid[r][c] is None:
                # Try horizontal answers for row r
                for word in horizontal_answers[r]:
                    if validate_word(grid, word, r, c, 'horizontal'):
                        new_state = fill_word(grid, word, r, c, 'horizontal')
                        successor_states.append(new_state)

                # Try vertical answers for column c
                for word in vertical_answers[c]:
                    if validate_word(grid, word, r, c, 'vertical'):
                        new_state = fill_word(grid, word, r, c, 'vertical')
                        successor_states.append(new_state)

    return successor_states
```

This function `get_successor_states` does the following:
1. Iterates over each cell in the grid.
2. Validates the possible horizontal and vertical words at each position using `validate_word`.
3. If valid, creates a new grid with the word filled in using `fill_word`.
4. Collects and returns all possible successor states.

This process should be repeated iteratively to explore all possible puzzle completions.

AutoToS prompt:

Please provide a goal test function in Python ensuring that all cells are not None, all words horizontally are matching at least one of the horizontal answers, and all words vertically match at least one of the vertical answers. The function should receive a state, horizontal_answers, and vertical_answers as before and return True for a goal state and False otherwise.

Model response:
To determine if the given state of the crossword puzzle is a goal state, we need to ensure the following conditions:
1. All cells in the grid are filled (not ‘None‘).
2. Every horizontal word in the grid matches one of the possible horizontal answers.
3. Every vertical word in the grid matches one of the possible vertical answers.

Let’s define this goal test function in Python:

```python
def is_goal_state(state, horizontal_answers, vertical_answers):
    def get_word(state, start_r, start_c, direction):
        word = []
        if direction == 'horizontal':
            while start_c < len(state[0]) and state[start_r][start_c]
                is not None:
                word.append(state[start_r][start_c])
                start_c += 1
        elif direction == 'vertical':
            while start_r < len(state) and state[start_r][start_c] is
                not None:
                word.append(state[start_r][start_c])
                start_r += 1
        return ''.join(word)

    def check_all_cells_filled(state):
        for row in state:
            for cell in row:
                if cell is None:
                    return False
        return True

    def check_words(state, answers, direction):
        n_rows, n_cols = len(state), len(state[0])
        if direction == 'horizontal':
            for r in range(n_rows):
                c = 0
                while c < n_cols:
                    if state[r][c] is not None:
                        word = get_word(state, r, c, direction)
                        if word not in answers[r]:
                            return False
                        c += len(word)
                    else:
                        c += 1
        elif direction == 'vertical':
            for c in range(n_cols):
                r = 0
                while r < n_rows:
                    if state[r][c] is not None:
                        word = get_word(state, r, c, direction)
                        if word not in answers[c]:
                            return False
                        r += len(word)
```

```
                    else:
                        r += 1
        return True

    return (check_all_cells_filled(state) and
            check_words(state, horizontal_answers, 'horizontal') and
            check_words(state, vertical_answers, 'vertical'))
```

This function `is_goal_state` performs the following steps:
1. `get_word` extracts a word from the grid starting at a specific position and extending in the given direction (`horizontal` or `vertical`).
2. `check_all_cells_filled` checks if all the cells in the grid are filled (none of them is `None`).
3. `check_words` validates that each horizontal and vertical word in the grid matches one of the given answers for the respective rows or columns.
4. `is_goal_state` combines these checks and returns `True` if the grid is fully filled and all words match the provided answers, otherwise returns `False`.

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 20, failure: 0, total expansions 4514757, total generated 16830232

Listing 41: With partial successor soundness test

AutoToS prompt:

The task is a 5x5 mini crosswords. A state is a 2D array representing the current puzzle state, where the initial grid is all "None". Note that some of the possible answers are not exactly 5 character long. Given an input of possible answers to horizontal clues and vertical clues, how is the successor state function defined? Please first think step by step. Then provide the successor state function in Python code.

The possible clues for each row and each column are given separately. Here is an example of possible horizontal and vertical clues:

```
horizontal_answers = [
        ["tasks", "goals", "plans", "agend", "chores", "works", "deeds", "items", "lists", "brief"],
        ["motor", "power", "drive", "diesel", "steam", "pumps", "crank", "gears", "turbn", "motor"],
        ["grand", "artsy", "showy", "ornate", "fancy", "vain", "proud", "vogue", "swank", "luxus"],
        ["venue", "salle", "forum", "atria", "lobby", "parls", "court", "malls", "mall", "lobby"],
        ["jeer", "scoff", "sleer", "deris", "sneer", "scorn", "derid", "gibes", "gibed", "flout"]
]
vertical_answers = [
        ["amass", "stack", "hoard", "pile", "store", "heaps", "massy", "gathe", "lumps", "mound"],
```

```
        ["nilga", "goral", "eland", "lepus", "gazal", "kudu", "oryx",
            "gnu", "imps", "carb"],
        ["scheme", "design", "ettle", "nettle", "sting", "wiles", "
            plans", "ideas", "plots", "cocks"],
        ["spout", "nosle", "snout", "mouth", "nostr", "ports", "inlet
            ", "vents", "outlt", "beaks"],
        ["drier", "arid", "sere", "parch", "dryer", "wring", "drear",
            "sear", "pall", "lack"]
]
```

where horizontal_answers is a list where element i is a list of possible answers to clue in row i, and vertical_answers is a list where element i is a list of possible answers to clue in column i.

Model response:
To generate the successor state function for a 5x5 mini crossword grid, you need to consider updating the state gradually by filling in words from the possible answers for horizontal and vertical clues. This function will return a list of valid successor states based on the next possible word for both horizontal and vertical clues.

When implementing this, you shall:
1. Define the initial empty 5x5 grid.
2. Iterate through each possible word for the horizontal clues and place them in the grid.
3. For each horizontal placement, iterate through each possible word for the vertical clues and place them in the grid.
4. Ensure that any new word placed must align and be valid with existing placements.

Here’s the Python function:

```python
def successor_state(state, horizontal_answers, vertical_answers):
    import copy

    def is_valid_state(state):
        for i in range(5):
            horizontal_word = ''.join([state[i][j] if state[i][j] is
                not None else '*' for j in range(5)])
            vertical_word = ''.join([state[j][i] if state[j][i] is not
                 None else '*' for j in range(5)])

            if horizontal_word.strip('*') and horizontal_word.strip
                ('*') not in valid_horizontal_words[i]:
                return False
            if vertical_word.strip('*') and vertical_word.strip('*')
                not in valid_vertical_words[i]:
                return False
        return True

    def generate_successor_states(state, possible_answers,
        is_horizontal):
        successors = []
        for i in range(5):
            for word in possible_answers[i]:
                new_state = copy.deepcopy(state)
                if is_horizontal:
                    if len(word) <= 5:
                        for j in range(len(word)):
                            new_state[i][j] = word[j]
                else:
                    if len(word) <= 5:
                        for j in range(len(word)):
                            new_state[j][i] = word[j]
```

```
                if is_valid_state(new_state):
                    successors.append(new_state)
        return successors

    valid_horizontal_words = [
        set(horizontal_answers[i]) for i in range(5)
    ]
    valid_vertical_words = [
        set(vertical_answers[i]) for i in range(5)
    ]

    current_turn_horizontal = all(state[i][j] is None for i in range(5) for j in range(5))
    if current_turn_horizontal:
        return generate_successor_states(state, horizontal_answers, True)
    else:
        return generate_successor_states(state, vertical_answers, False)
```

Explanation
- **`is_valid_state`**: This validates if the current state of the grid forms valid partial words horizontally and vertically.
- **`generate_successor_states`**: This generates successor states by attempting to place each possible word in the grid for either horizontal or vertical positions and returns valid new states.
- **Initial Check**: The function determines whether to place horizontal words first by checking if the state is entirely empty.

This function will proceed to fill the state with valid words step-by-step for both horizontal and vertical clues while ensuring intermediate states are valid.

AutoToS prompt:

Please provide a goal test function in Python ensuring that all cells are not None, all words horizontally are matching at least one of the horizontal answers, and all words vertically match at least one of the vertical answers. The function should receive a state, horizontal_answers, and vertical_answers as before and return True for a goal state and False otherwise.

Model response:
To implement a goal test function in Python, you need to ensure that:
1. All cells in the state are filled (i.e., not `None`).
2. Each horizontal word matches one of the horizontal answers.
3. Each vertical word matches one of the vertical answers.

Here is the function to achieve this:

```python
def is_goal_state(state, horizontal_answers, vertical_answers):
    def all_filled(state):
        for row in state:
            if any(cell is None for cell in row):
                return False
        return True

    def all_horizontal_words_valid(state, horizontal_answers):
        for i, row in enumerate(state):
            word = ''.join(row)
            if word not in horizontal_answers[i]:
                return False
```

```
        return True

    def all_vertical_words_valid(state, vertical_answers):
        for i in range(5):
            word = ''.join(state[j][i] for j in range(5))
            if word not in vertical_answers[i]:
                return False
        return True

    if not all_filled(state):
        return False
    if not all_horizontal_words_valid(state, horizontal_answers):
        return False
    if not all_vertical_words_valid(state, vertical_answers):
        return False

    return True
```

Explanation
1. **`all_filled(state)`**: Checks that every cell in the 5x5 grid is filled (no `None` values).
2. **`all_horizontal_words_valid(state, horizontal_answers)`**: Checks that every horizontal word formed in the state matches one of the allowed horizontal answers.
3. **`all_vertical_words_valid(state, vertical_answers)`**: Checks that every vertical word formed in the state matches one of the allowed vertical answers.

The main function `is_goal_state` aggregates these checks, returning `True` only if all conditions are satisfied, otherwise `False`.

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
Successor state has the same number of filled cells as the parent state.
Let's think step by step. First, think what you did wrong.
Then, think of in what ways successor state should be different from the parent state.
Then, provide the complete Python code for the revised successor function that returns a list of successor states.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: [[None, None, None, None, None], [None, None, None, None, None], [None, None, None, None, None], [None, None, None, None, None], [None, None, None, None, None]]
Example wrong successor state: [[None, None, None, None, None], [None, None, None, None, None], [None, None, None, None, None], [None, None, None, None, None], [None, None, None, None, None]]
Horizontal clues: [['tasks', 'goals', 'plans', 'agend', 'chores', 'works', 'deeds', 'items', 'lists', 'brief'], ['motor', 'power', 'drive', 'diesel', 'steam', 'pumps', 'crank', 'gears', 'turbn', 'motor'], ['grand', 'artsy', 'showy', 'ornate', 'fancy', 'vain', 'proud', 'vogue', 'swank', 'luxus'], ['venue', 'salle', 'forum', 'atria', 'lobby', 'parls', 'court', 'malls', 'mall', 'lobby'], ['jeer', 'scoff', 'sleer', 'deris', 'sneer', 'scorn', 'derid', 'gibes', 'gibed', 'flout']]
Vertical clues: [['amass', 'stack', 'hoard', 'pile', 'store', 'heaps', 'massy', 'gathe', 'lumps', 'mound'], ['nilga', 'goral', 'eland', 'lepus', 'gazal', 'kudu', 'oryx', 'gnu', 'imps', 'carb'], ['scheme

', 'design', 'ettle', 'nettle', 'sting', 'wiles', 'plans', 'ideas
', 'plots', 'cocks'], ['spout', 'nosle', 'snout', 'mouth', 'nostr
', 'ports', 'inlet', 'vents', 'outlt', 'beaks'], ['drier', 'arid',
'sere', 'parch', 'dryer', 'wring', 'drear', 'sear', 'pall', 'lack
']]

Model response:
To address the issues with the original approach, let's think step-by-step:

1. **Identify the Mistakes**:
   - The original function attempted to generate successor states directly without guaranteeing any new cells were filled. Therefore, it could return states identical to the input.
   - It did not handle the transition from placing horizontal words to vertical words clearly, which must happen sequentially.

2. **Determine the Required Changes**:
   - A successor state should differ from the parent state by having at least one additional word placed in the grid.
   - Generate valid successor states by filling the grid according to the current phase (placing horizontal or vertical words). Start with horizontal placement and switch to vertical once all rows are attempted.

3. **Revised Successor Function**:
   - The function must clearly track the filling process and ensure progression, creating new states with progressively filled cells.

Here is the revised successor function which ensures that the successor state has more filled cells than the parent state and handles the transitions properly:

```python
def successor_state(state, horizontal_answers, vertical_answers):
    import copy

    def is_valid_state(state):
        for i in range(5):
            horizontal_word = ''.join([state[i][j] if state[i][j] is
                not None else '*' for j in range(5)])
            vertical_word = ''.join([state[j][i] if state[j][i] is not
                None else '*' for j in range(5)])

            if horizontal_word.strip('*') and horizontal_word.strip
                ('*') not in valid_horizontal_words[i]:
                return False
            if vertical_word.strip('*') and vertical_word.strip('*')
                not in valid_vertical_words[i]:
                return False
        return True

    def generate_horizontal_successors(state):
        successors = []
        for i in range(5):
            if all(state[i][j] is None for j in range(5)):  # Find the
                first completely empty row
                for word in horizontal_answers[i]:
                    if len(word) <= 5:
                        new_state = copy.deepcopy(state)
                        for j in range(len(word)):
                            new_state[i][j] = word[j]

                        if is_valid_state(new_state):
```

```
                        successors.append(new_state)
                return successors
        return successors

    def generate_vertical_successors(state):
        successors = []
        for i in range(5):
            if all(state[j][i] is None for j in range(5)):  # Find the
                 first completely empty column
                for word in vertical_answers[i]:
                    if len(word) <= 5:
                        new_state = copy.deepcopy(state)
                        for j in range(len(word)):
                            new_state[j][i] = word[j]

                        if is_valid_state(new_state):
                            successors.append(new_state)
                return successors
        return successors

    valid_horizontal_words = [
        set(horizontal_answers[i]) for i in range(5)
    ]
    valid_vertical_words = [
        set(vertical_answers[i]) for i in range(5)
    ]

    # Determine if the grid has completely empty rows or columns to
       decide mode
    horizontal_successors = generate_horizontal_successors(state)
    if horizontal_successors:
        return horizontal_successors

    vertical_successors = generate_vertical_successors(state)
    if vertical_successors:
        return vertical_successors

    return []

```

Explanation in Detail
1. **`is_valid_state`**: Ensures that horizontal and vertical partial words formed in the grid so far are valid.
2. **`generate_horizontal_successors`**: Generates valid successor states by attempting to fill the first completely empty row with each possible horizontal word from the clues.
3. **`generate_vertical_successors`**: Generates valid successor states by attempting to fill the first completely empty column with each possible vertical word from the clues.
4. **Determine Mode**: The function first tries to generate successors by filling horizontal words. If this is not possible (all rows are filled), it generates successors by filling vertical words.
5. **Sequential Progression**: Ensures that once all horizontal placements are attempted, it will move to vertical placements.

This approach ensures a valid progression from the parent to successor states, and that each new state has more filled cells than its predecessor.

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 2

```
AutoToS prompt:
Successor function when run on the state [[None, None, 'e', None, None
    ], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None
    , None, 'l', None, None], [None, None, 'e', None, None]] failed to
     produce all successors.
Missing successors are: [[['a', 'g', 'e', 'n', 'd'], ['m', 'o', 't', '
    o', 'r'], [None, None, 't', None, None], [None, None, 'l', None,
    None], [None, None, 'e', None, None]], [['d', 'e', 'e', 'd', 's'],
     ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None,
    None, 'l', None, None], [None, None, 'e', None, None]], [['i', 't
    ', 'e', 'm', 's'], ['m', 'o', 't', 'o', 'r'], [None, None, 't',
    None, None], [None, None, 'l', None, None], [None, None, 'e', None
    , None]], [[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r
    '], ['a', 'r', 't', 's', 'y'], [None, None, 'l', None, None], [
    None, None, 'e', None, None]], [[None, None, 'e', None, None], ['m
    ', 'o', 't', 'o', 'r'], [None, None, 't', None, None], ['s', 'a',
    'l', 'l', 'e'], [None, None, 'e', None, None]], [[None, None, 'e',
     None, None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None,
    None], ['m', 'a', 'l', 'l', 's'], [None, None, 'e', None, None]],
    [[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None,
    None, 't', None, None], [None, None, 'l', None, None], ['s', 'l',
    'e', 'e', 'r']], [[None, None, 'e', None, None], ['m', 'o', 't', '
    o', 'r'], [None, None, 't', None, None], [None, None, 'l', None,
    None], ['s', 'n', 'e', 'e', 'r']], [['a', None, 'e', None, None],
    ['m', 'o', 't', 'o', 'r'], ['a', None, 't', None, None], ['s',
    None, 'l', None, None], ['s', None, 'e', None, None]], [[None, 'g
    ', 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None, 'r', 't',
    None, None], [None, 'a', 'l', None, None], [None, 'l', 'e', None,
    None]], [[None, None, 'e', 'n', None], ['m', 'o', 't', 'o', 'r'],
    [None, None, 't', 's', None], [None, None, 'l', 'l', None], [None,
     None, 'e', 'e', None]], [[None, None, 'e', 'm', None], ['m', 'o',
     't', 'o', 'r'], [None, None, 't', 'u', None], [None, None, 'l', '
    t', None], [None, None, 'e', 'h', None]], [[None, None, 'e', 'n',
    None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', 's', None], [
    None, None, 'l', 't', None], [None, None, 'e', 'r', None]], [[None
    , None, 'e', 'p', None], ['m', 'o', 't', 'o', 'r'], [None, None, '
    t', 'r', None], [None, None, 'l', 't', None], [None, None, 'e', 's
    ', None]], [[None, None, 'e', None, 'd'], ['m', 'o', 't', 'o', 'r
    '], [None, None, 't', None, 'i'], [None, None, 'l', None, 'e'], [
    None, None, 'e', None, 'r']], [[None, None, 'e', None, 'd'], ['m',
     'o', 't', 'o', 'r'], [None, None, 't', None, 'y'], [None, None, '
    l', None, 'e'], [None, None, 'e', None, 'r']], [[None, None, 'e',
    None, 'w'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, 'i
    '], [None, None, 'l', None, 'n'], [None, None, 'e', None, 'g']],
    [[None, None, 'e', None, 'd'], ['m', 'o', 't', 'o', 'r'], [None,
    None, 't', None, 'e'], [None, None, 'l', None, 'a'], [None, None,
    'e', None, 'r']]]
First think step by step why the successor function failed to produce
    all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.
Horizontal clues: [['tasks', 'goals', 'plans', 'agend', 'chores', '
    works', 'deeds', 'items', 'lists', 'brief'], ['motor', 'power', '
    drive', 'diesel', 'steam', 'pumps', 'crank', 'gears', 'turbn', '
    motor'], ['grand', 'artsy', 'showy', 'ornate', 'fancy', 'vain', '
    proud', 'vogue', 'swank', 'luxus'], ['venue', 'salle', 'forum', '
    atria', 'lobby', 'parls', 'court', 'malls', 'mall', 'lobby'], ['
    jeer', 'scoff', 'sleer', 'deris', 'sneer', 'scorn', 'derid', '
    gibes', 'gibed', 'flout']]
Vertical clues: [['amass', 'stack', 'hoard', 'pile', 'store', 'heaps',
     'massy', 'gathe', 'lumps', 'mound'], ['nilga', 'goral', 'eland',
    'lepus', 'gazal', 'kudu', 'oryx', 'gnu', 'imps', 'carb'], ['scheme
    ', 'design', 'ettle', 'nettle', 'sting', 'wiles', 'plans', 'ideas
    ', 'plots', 'cocks'], ['spout', 'nosle', 'snout', 'mouth', 'nostr
```

', 'ports', 'inlet', 'vents', 'outlt', 'beaks'], ['drier', 'arid', 'sere', 'parch', 'dryer', 'wring', 'drear', 'sear', 'pall', 'lack']]

Model response:
To understand why the successor function failed to produce all successors, we need to reconsider the scope and exact procedure of generating successors. The function must generate valid states by extending the current state through placing valid words in the next available row or column, one at a time.

Here are the key steps to fix the function:
1. Ensure the function generates successors by filling one empty row or column at a time.
2. Validate that placing a word in a row or column doesn't invalidate the grid.
3. Differentiate between generating horizontal or vertical successors based on the current state.

Revised Plan
1. Identify the first unfilled row for horizontal successors, and the first unfilled column for vertical successors.
2. Generate all possible states by filling this row or column with every possible word from the respective answers.
3. Each state must align with previously filled parts of the grid.

Revised Successor Function
Here is the revised code with enhanced logic to ensure all valid successors are generated:

```python
def successor_state(state, horizontal_answers, vertical_answers):
    import copy

    def is_valid_state(state):
        for i in range(5):
            horizontal_word = ''.join([state[i][j] if state[i][j] is
                not None else '*' for j in range(5)])
            vertical_word = ''.join([state[j][i] if state[j][i] is not
                 None else '*' for j in range(5)])

            if horizontal_word.strip('*') and horizontal_word.strip
                ('*') not in valid_horizontal_words[i]:
                return False
            if vertical_word.strip('*') and vertical_word.strip('*')
                not in valid_vertical_words[i]:
                return False
        return True

    def generate_horizontal_successors(state):
        successors = []
        for i in range(5):
            if any(state[i][j] is None for j in range(5)):
                for word in horizontal_answers[i]:
                    if len(word) <= 5:
                        new_state = copy.deepcopy(state)
                        can_fit = True
                        for j in range(len(word)):
                            if state[i][j] is not None and state[i][j]
                                 != word[j]:
                                can_fit = False
                                break
                            new_state[i][j] = word[j]
                        if can_fit and is_valid_state(new_state):
                            successors.append(new_state)
```

```
                break
        return successors

    def generate_vertical_successors(state):
        successors = []
        for i in range(5):
            if any(state[j][i] is None for j in range(5)):
                for word in vertical_answers[i]:
                    if len(word) <= 5:
                        new_state = copy.deepcopy(state)
                        can_fit = True
                        for j in range(len(word)):
                            if state[j][i] is not None and state[j][i]
                                 != word[j]:
                                can_fit = False
                                break
                            new_state[j][i] = word[j]
                        if can_fit and is_valid_state(new_state):
                            successors.append(new_state)
                break
        return successors

    valid_horizontal_words = [
        set(horizontal_answers[i]) for i in range(5)
    ]
    valid_vertical_words = [
        set(vertical_answers[i]) for i in range(5)
    ]

    horizontal_successors = generate_horizontal_successors(state)
    if horizontal_successors:
        return horizontal_successors

    vertical_successors = generate_vertical_successors(state)
    if vertical_successors:
        return vertical_successors

    return []
```

Explanation:
1. **`is_valid_state`**: Ensures partial words formed so far match the clues.
2. **`generate_horizontal_successors`**:
   - Finds the first row with any `None`.
   - Tries to place each possible word from the horizontal answers in that row.
   - Checks if the word aligns with any pre-filled cells and if the resulting state is valid.
3. **`generate_vertical_successors`**:
   - Similar logic to horizontal successors but for columns.
4. **Generation Control**:
   - Tries to generate horizontal successors first. If none are found (i.e., all rows are filled), then it tries vertical successors.

Conclusion:
This revised function ensures that the successor states progressively fill unknown cells, thereby generating complete sets of valid successor states.

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 3

```
AutoToS prompt:
Successor function when run on the state [[None, None, ’e’, None, None
    ], [’m’, ’o’, ’t’, ’o’, ’r’], [None, None, ’t’, None, None], [None
    , None, ’l’, None, None], [None, None, ’e’, None, None]] failed to
     produce all successors.
Missing successors are: [[[’a’, ’g’, ’e’, ’n’, ’d’], [’m’, ’o’, ’t’, ’
    o’, ’r’], [None, None, ’t’, None, None], [None, None, ’l’, None,
    None], [None, None, ’e’, None, None]], [[’d’, ’e’, ’e’, ’d’, ’s’],
     [’m’, ’o’, ’t’, ’o’, ’r’], [None, None, ’t’, None, None], [None,
    None, ’l’, None, None], [None, None, ’e’, None, None]], [[’i’, ’t
    ’, ’e’, ’m’, ’s’], [’m’, ’o’, ’t’, ’o’, ’r’], [None, None, ’t’,
    None, None], [None, None, ’l’, None, None], [None, None, ’e’, None
    , None]], [[None, None, ’e’, None, None], [’m’, ’o’, ’t’, ’o’, ’r
    ’], [’a’, ’r’, ’t’, ’s’, ’y’], [None, None, ’l’, None, None], [
    None, None, ’e’, None, None]], [[None, None, ’e’, None, None], [’m
    ’, ’o’, ’t’, ’o’, ’r’], [None, None, ’t’, None, None], [’s’, ’a’,
    ’l’, ’l’, ’e’], [None, None, ’e’, None, None]], [[None, None, ’e’,
     None, None], [’m’, ’o’, ’t’, ’o’, ’r’], [None, None, ’t’, None,
    None], [’m’, ’a’, ’l’, ’l’, ’s’], [None, None, ’e’, None, None]],
    [[None, None, ’e’, None, None], [’m’, ’o’, ’t’, ’o’, ’r’], [None,
    None, ’t’, None, None], [None, None, ’l’, None, None], [’s’, ’l’,
    ’e’, ’e’, ’r’]], [[None, None, ’e’, None, None], [’m’, ’o’, ’t’, ’
    o’, ’r’], [None, None, ’t’, None, None], [None, None, ’l’, None,
    None], [’s’, ’n’, ’e’, ’e’, ’r’]], [[’a’, None, ’e’, None, None],
    [’m’, ’o’, ’t’, ’o’, ’r’], [’a’, None, ’t’, None, None], [’s’,
    None, ’l’, None, None], [’s’, None, ’e’, None, None]], [[None, ’g
    ’, ’e’, None, None], [’m’, ’o’, ’t’, ’o’, ’r’], [None, ’r’, ’t’,
    None, None], [None, ’a’, ’l’, None, None], [None, ’l’, ’e’, None,
    None]], [[None, None, ’e’, ’n’, None], [’m’, ’o’, ’t’, ’o’, ’r’],
    [None, None, ’t’, ’s’, None], [None, None, ’l’, ’l’, None], [None,
     None, ’e’, ’e’, None]], [[None, None, ’e’, ’m’, None], [’m’, ’o’,
     ’t’, ’o’, ’r’], [None, None, ’t’, ’u’, None], [None, None, ’l’, ’
    t’, None], [None, None, ’e’, ’h’, None]], [[None, None, ’e’, ’n’,
    None], [’m’, ’o’, ’t’, ’o’, ’r’], [None, None, ’t’, ’s’, None], [
    None, None, ’l’, ’t’, None], [None, None, ’e’, ’r’, None]], [[None
    , None, ’e’, ’p’, None], [’m’, ’o’, ’t’, ’o’, ’r’], [None, None, ’
    t’, ’r’, None], [None, None, ’l’, ’t’, None], [None, None, ’e’, ’s
    ’, None]], [[None, None, ’e’, None, ’d’], [’m’, ’o’, ’t’, ’o’, ’r
    ’], [None, None, ’t’, None, ’i’], [None, None, ’l’, None, ’e’], [
    None, None, ’e’, None, ’r’]], [[None, None, ’e’, None, ’d’], [’m’,
     ’o’, ’t’, ’o’, ’r’], [None, None, ’t’, None, ’y’], [None, None, ’
    l’, None, ’e’], [None, None, ’e’, None, ’r’]], [[None, None, ’e’,
    None, ’w’], [’m’, ’o’, ’t’, ’o’, ’r’], [None, None, ’t’, None, ’i
    ’], [None, None, ’l’, None, ’n’], [None, None, ’e’, None, ’g’]],
    [[None, None, ’e’, None, ’d’], [’m’, ’o’, ’t’, ’o’, ’r’], [None,
    None, ’t’, None, ’e’], [None, None, ’l’, None, ’a’], [None, None,
    ’e’, None, ’r’]]]
First think step by step why the successor function failed to produce
    all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.
Horizontal clues: [[’tasks’, ’goals’, ’plans’, ’agend’, ’chores’, ’
    works’, ’deeds’, ’items’, ’lists’, ’brief’], [’motor’, ’power’, ’
    drive’, ’diesel’, ’steam’, ’pumps’, ’crank’, ’gears’, ’turbn’, ’
    motor’], [’grand’, ’artsy’, ’showy’, ’ornate’, ’fancy’, ’vain’, ’
    proud’, ’vogue’, ’swank’, ’luxus’], [’venue’, ’salle’, ’forum’, ’
    atria’, ’lobby’, ’parls’, ’court’, ’malls’, ’mall’, ’lobby’], [’
    jeer’, ’scoff’, ’sleer’, ’deris’, ’sneer’, ’scorn’, ’derid’, ’
    gibes’, ’gibed’, ’flout’]]
Vertical clues: [[’amass’, ’stack’, ’hoard’, ’pile’, ’store’, ’heaps’,
     ’massy’, ’gathe’, ’lumps’, ’mound’], [’nilga’, ’goral’, ’eland’,
    ’lepus’, ’gazal’, ’kudu’, ’oryx’, ’gnu’, ’imps’, ’carb’], [’scheme
    ’, ’design’, ’ettle’, ’nettle’, ’sting’, ’wiles’, ’plans’, ’ideas
    ’, ’plots’, ’cocks’], [’spout’, ’nosle’, ’snout’, ’mouth’, ’nostr
```

', 'ports', 'inlet', 'vents', 'outlt', 'beaks'], ['drier', 'arid',
'sere', 'parch', 'dryer', 'wring', 'drear', 'sear', 'pall', 'lack
']]

Model response:
To understand why the previous successor function failed, let's carefully examine our logic:

1. **Current Problems**:
   - The previous function did not correctly identify and handle rows and columns that were partially filled.
   - The iteration and placement logic were not robust enough to cover all cases of partial fillings.

2. **Required Corrections**:
   - Identify both fully and partially filled rows/columns correctly.
   - Ensure the function correctly places words by matching partially filled cells.
   - Generate all valid successors by placing words in the next available row or column one at a time.

Revised Successor Function
Here is the corrected and more comprehensive function:

```python
def successor_state(state, horizontal_answers, vertical_answers):
    import copy

    def is_valid_state(state):
        for i in range(5):
            # Check each horizontal word
            horizontal_word = ''.join([state[i][j] if state[i][j] is
                not None else '*' for j in range(5)])
            if '*' in horizontal_word:
                horizontal_word = horizontal_word.split('*')[0]
            if horizontal_word and horizontal_word not in
                valid_horizontal_words[i]:
                return False

            # Check each vertical word
            vertical_word = ''.join([state[j][i] if state[j][i] is not
                 None else '*' for j in range(5)])
            if '*' in vertical_word:
                vertical_word = vertical_word.split('*')[0]
            if vertical_word and vertical_word not in
                valid_vertical_words[i]:
                return False
        return True

    def generate_successors(state, possible_answers, is_horizontal):
        successors = []
        for i in range(5):
            indices_to_fill = [None] * 5
            valid_row = True
            for j in range(5 if is_horizontal else 5):
                if is_horizontal:
                    if state[i][j] is not None:
                        indices_to_fill[j] = state[i][j]
                else:
                    if state[j][i] is not None:
                        indices_to_fill[j] = state[j][i]

            if all(ch is not None for ch in indices_to_fill):  # Skip
                fully filled rows/columns
                continue
```

```
            for word in possible_answers[i]:
                new_state = copy.deepcopy(state)
                can_fit = True
                for j, ch in enumerate(word):
                    if indices_to_fill[j] is not None and indices_to_fill[j] != ch:
                        can_fit = False
                        break
                    if is_horizontal:
                        new_state[i][j] = ch
                    else:
                        new_state[j][i] = ch

                if can_fit and is_valid_state(new_state):
                    successors.append(new_state)
            if successors:  # If we're able to generate at least one valid successor, stop and return
                return successors
        return []

    valid_horizontal_words = [
        set(horizontal_answers[i]) for i in range(5)
    ]
    valid_vertical_words = [
        set(vertical_answers[i]) for i in range(5)
    ]

    # Try to generate horizontal successors first, if no valid horizontal successors, then vertical.
    horizontal_successors = generate_successors(state, horizontal_answers, True)
    if horizontal_successors:
        return horizontal_successors

    vertical_successors = generate_successors(state, vertical_answers, False)
    if vertical_successors:
        return vertical_successors

    return []
```

Explanation:
- **`is_valid_state(state)`**: Checks partial words horizontally and vertically to ensure they are valid sketched words.
    - Horizontal and vertical partial word checks include splitting by '*' to handle partially filled cells efficiently.
- **`generate_successors(state, possible_answers, is_horizontal)`**:
    - Determines whether to generate horizontal or vertical successors.
    - Iterates through each row/column and attempts to place possible words.
    - Ensures any new word placement aligns with the already placed characters.
    - Generates new states only if valid placements align with the current grid's state.
    - Stops and returns valid successors as soon as they are found, ensuring efficiency.

This function ensures that all possibilities are considered, and all valid successors are generated by properly extending the current state step-by-step.
Successor Iteration 4

AutoToS prompt:
The following exception was produced when testing the successor function:
  File "<string>", line 41, in generate_successors

list index out of range. Please fix this exception in the successor function.
Model response:
It looks like the function may be attempting to access indices that are out of bounds due to incorrect logic handling both horizontal and vertical expansions separately. This mistake usually happens when the function tries to process partially filled rows or columns with mismatched lengths.

Solution:
1. Ensure correct indexing within the bounds of the 5x5 grid.
2. Split and check each potential word placement to match the exact indices in the grid.
3. Fix any off-by-one errors by carefully managing the row and column limits.

Here is the revised function:

```python
def successor_state(state, horizontal_answers, vertical_answers):
    import copy

    def is_valid_state(state):
        for i in range(5):
            # Check each horizontal word
            horizontal_word = ''.join([state[i][j] if state[i][j] is
                not None else '*' for j in range(5)])
            if '*' in horizontal_word:
                horizontal_word = horizontal_word.split('*')[0]
            if horizontal_word and horizontal_word not in
                valid_horizontal_words[i]:
                return False

            # Check each vertical word
            vertical_word = ''.join([state[j][i] if state[j][i] is not
                 None else '*' for j in range(5)])
            if '*' in vertical_word:
                vertical_word = vertical_word.split('*')[0]
            if vertical_word and vertical_word not in
                valid_vertical_words[i]:
                return False
        return True

    def generate_successors(state, possible_answers, is_horizontal):
        successors = []
        for i in range(5):
            if is_horizontal:
                indices_to_fill = [state[i][j] for j in range(5)]
            else:
                indices_to_fill = [state[j][i] for j in range(5)]

            if all(ch is not None for ch in indices_to_fill):  # Skip
                fully filled rows/columns
                continue

            for word in possible_answers[i]:
                new_state = copy.deepcopy(state)
                can_fit = True

                for j, ch in enumerate(word):
```

```
                    if (is_horizontal and state[i][j] is not None and
                        state[i][j] != ch) or (not is_horizontal and
                        state[j][i] is not None and state[j][i] != ch)
                        :
                        can_fit = False
                        break
                    if is_horizontal:
                        new_state[i][j] = ch
                    else:
                        new_state[j][i] = ch

                if can_fit and is_valid_state(new_state):
                    successors.append(new_state)

            # When an unfilled row/column is processed, attempt to
                return as soon as successors are found
            if successors:
                return successors

        return []

    valid_horizontal_words = [set(horizontal_answers[i]) for i in
        range(5)]
    valid_vertical_words = [set(vertical_answers[i]) for i in range(5)
        ]

    # Check if we should generate horizontal or vertical successors
        based on the current fill status.
    horizontal_successors = generate_successors(state,
        horizontal_answers, True)
    if horizontal_successors:
        return horizontal_successors

    vertical_successors = generate_successors(state, vertical_answers,
         False)
    if vertical_successors:
        return vertical_successors

    return []
```

Explanation:

1. **`is_valid_state(state)`**: Checks the grid to ensure partial and completed words are valid based on the provided answers. It handles both horizontal and vertical words using consistent bounds .
2. **`generate_successors(state, possible_answers, is_horizontal)`**:
    - Determines whether to fill rows (horizontal) or columns (vertical).
    - Ensures all indices accessed are within bounds (0 to 4) for the 5x5 grid.
    - Manages potential word placements and checks if they align with partially filled states.
    - Ensures any new state is valid before adding to the successors list.

This function maintains better indexing logic and handles both row and column expansions appropriately, preventing the index out-of-range error.
Successor Iteration 5

AutoToS prompt:
The following exception was produced when testing the successor function:
  File "<string>", line 37, in generate_successors

list index out of range. Please fix this exception in the successor function.
Model response:
To fix the "list index out of range" error, we need to carefully handle the bounds and ensure we don’t access indices outside of the 5x5 grid. Let’s refine the logic to ensure all operations are within valid indices.

Detailed Steps:

1. **Ensure correct bounds checking**: Make sure we are accessing only within the valid 5x5 grid.
2. **Correctly handle partially filled rows/columns**: Ensure methods for checking and placing words align with the grid’s state without exceeding boundaries.
3. **Refine logic to prevent off-by-one errors**.

Here’s the improved function checking for bounds explicitly:

```python
def successor_state(state, horizontal_answers, vertical_answers):
    import copy

    def is_valid_state(state):
        for i in range(5):
            # Check each horizontal word
            horizontal_word = ''.join([state[i][j] if state[i][j] is
                not None else '*' for j in range(5)])
            if '*' in horizontal_word:
                fragments = horizontal_word.split('*')
                horizontal_word = fragments[0] if fragments[0] else (
                    fragments[1] if len(fragments) > 1 else '')
            if horizontal_word and horizontal_word not in
                valid_horizontal_words[i]:
                return False

            # Check each vertical word
            vertical_word = ''.join([state[j][i] if state[j][i] is not
                 None else '*' for j in range(5)])
            if '*' in vertical_word:
                fragments = vertical_word.split('*')
                vertical_word = fragments[0] if fragments[0] else (
                    fragments[1] if len(fragments) > 1 else '')
            if vertical_word and vertical_word not in
                valid_vertical_words[i]:
                return False
        return True

    def generate_successors(state, possible_answers, is_horizontal):
        successors = []
        for i in range(5):
            indices_to_fill = []
            valid_row = True
            for j in range(5):
                if is_horizontal:
                    if state[i][j] is not None:
                        indices_to_fill.append(state[i][j])
                    else:
                        indices_to_fill.append(None)
                else:
                    if state[j][i] is not None:
                        indices_to_fill.append(state[j][i])
                    else:
                        indices_to_fill.append(None)
```

```
            if all(ch is not None for ch in indices_to_fill):  # Skip
                fully filled rows/columns
                continue

            for word in possible_answers[i]:
                new_state = copy.deepcopy(state)
                can_fit = True
                for j, ch in enumerate(word):
                    if indices_to_fill[j] is not None and
                        indices_to_fill[j] != ch:
                        can_fit = False
                        break
                    if is_horizontal:
                        new_state[i][j] = ch
                    else:
                        new_state[j][i] = ch
                if can_fit and is_valid_state(new_state):
                    successors.append(new_state)

            if successors:
                return successors  # Return after first successful row
                    /column

        return []

    valid_horizontal_words = [set(horizontal_answers[i]) for i in
        range(5)]
    valid_vertical_words = [set(vertical_answers[i]) for i in range(5)
        ]

    # Try to generate horizontal successors first, if no valid
        horizontal successors, then vertical.
    horizontal_successors = generate_successors(state,
        horizontal_answers, True)
    if horizontal_successors:
        return horizontal_successors

    vertical_successors = generate_successors(state, vertical_answers,
         False)
    if vertical_successors:
        return vertical_successors

    return []
```

Explanation:
- **Index Safety**: Managed indices through a bounded loop, ensuring any list access is within bounds (0 to 4).
- **Word Placement**: Added word placement logic that respects the filled state of each cell.
- **Validation Checks**: Included checks to ensure any modification stays within valid rows/columns, making sure the state update is safe.

This approach should prevent "list index out of range" errors by carefully controlling index access within the 5x5 limits.

Successor Iteration 6

AutoToS prompt:
The following exception was produced when testing the successor function:
  File "<string>", line 47, in generate_successors

list index out of range. Please fix this exception in the successor function.
Model response:
The "list index out of range" error typically arises from trying to access or iterate beyond the bounds of the list. I will carefully review the logic to ensure we access indices properly within the 5x5 grid.

Let's refine the function to prevent such errors:

1. We need to confirm that for each word we try to place horizontally or vertically, it doesn't exceed the bounds of the 5x5 grid.
2. Fix any indexing and bounds checking errors to make sure we stay within the correct range.

Here is the revised function:

```python
def successor_state(state, horizontal_answers, vertical_answers):
    import copy

    def is_valid_state(state):
        for i in range(5):
            # Check each horizontal word
            horizontal_word = ''.join([state[i][j] if state[i][j] is
                not None else '*' for j in range(5)])
            if '*' in horizontal_word:
                fragments = horizontal_word.split('*')
                horizontal_word = fragments[0] if fragments[0] else (
                    fragments[1] if len(fragments) > 1 else '')
            if horizontal_word and horizontal_word not in
                valid_horizontal_words[i]:
                return False

            # Check each vertical word
            vertical_word = ''.join([state[j][i] if state[j][i] is not
                 None else '*' for j in range(5)])
            if '*' in vertical_word:
                fragments = vertical_word.split('*')
                vertical_word = fragments[0] if fragments[0] else (
                    fragments[1] if len(fragments) > 1 else '')
            if vertical_word and vertical_word not in
                valid_vertical_words[i]:
                return False
        return True

    def generate_successors(state, possible_answers, is_horizontal):
        successors = []
        for i in range(5):
            if is_horizontal:
                current_line = [state[i][j] for j in range(5)]
            else:
                current_line = [state[j][i] for j in range(5)]

            if all(ch is not None for ch in current_line):  # Skip
                fully filled rows/columns
                continue

            for word in possible_answers[i]:
                if len(word) > 5:
                    continue
                new_state = copy.deepcopy(state)
                can_fit = True
                for j in range(5):
                    ch = word[j] if j < len(word) else None
```

```
                    if current_line[j] is not None and current_line[j] != ch:
                        can_fit = False
                        break
                    if is_horizontal and ch is not None:
                        new_state[i][j] = ch
                    elif not is_horizontal and ch is not None:
                        new_state[j][i] = ch
                if can_fit and is_valid_state(new_state):
                    successors.append(new_state)

            if successors:
                return successors  # Return successors after processing the first row/column with empty cells

        return successors

    valid_horizontal_words = [set(horizontal_answers[i]) for i in range(5)]
    valid_vertical_words = [set(vertical_answers[i]) for i in range(5)]

    # Try to generate horizontal successors first, if no valid horizontal successors, then vertical.
    horizontal_successors = generate_successors(state, horizontal_answers, True)
    if horizontal_successors:
        return horizontal_successors

    vertical_successors = generate_successors(state, vertical_answers, False)
    if vertical_successors:
        return vertical_successors

    return []

```

Explanation:
1. **`is_valid_state(state)`**: Checks whether horizontal and vertical words formed match the given clues, managing splitting by `'*'` properly.
2. **`generate_successors(state, possible_answers, is_horizontal)`**:
   - It identifies whether to work on rows (horizontal) or columns (vertical).
   - It constructs `current_line` from the grid and guarantees it won't exceed the bounds (`len(word) <= 5`).
   - Logic now carefully ensures word placement does not extend beyond the 5x5 range.
   - After processing the first row/column with empty cells, it returns the generated successors.

This refined approach should manage indices within the expected range of the grid and prevent "list index out of range" errors.
Successor Iteration 7

AutoToS prompt:
Successor state has the same number of filled cells as the parent state.
Let's think step by step. First, think what you did wrong.
Then, think of in what ways successor state should be different from the parent state.
Then, provide the complete Python code for the revised successor function that returns a list of successor states.

Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: [[None, None, None, None, None], [None, None, None, None, None], ['l', 'u', 'x', 'u', 's'], ['l', 'o', 'b', 'b', 'y'], ['j', 'e', 'e', 'r', None]]
Example wrong successor state: [[None, None, None, None, None], [None, None, None, None, None], ['l', 'u', 'x', 'u', 's'], ['l', 'o', 'b', 'b', 'y'], ['j', 'e', 'e', 'r', None]]
Horizontal clues: [['tasks', 'goals', 'plans', 'agend', 'chores', 'works', 'deeds', 'items', 'lists', 'brief'], ['motor', 'power', 'drive', 'diesel', 'steam', 'pumps', 'crank', 'gears', 'turbn', 'motor'], ['grand', 'artsy', 'showy', 'ornate', 'fancy', 'vain', 'proud', 'vogue', 'swank', 'luxus'], ['venue', 'salle', 'forum', 'atria', 'lobby', 'parls', 'court', 'malls', 'mall', 'lobby'], ['jeer', 'scoff', 'sleer', 'deris', 'sneer', 'scorn', 'derid', 'gibes', 'gibed', 'flout']]
Vertical clues: [['amass', 'stack', 'hoard', 'pile', 'store', 'heaps', 'massy', 'gathe', 'lumps', 'mound'], ['nilga', 'goral', 'eland', 'lepus', 'gazal', 'kudu', 'oryx', 'gnu', 'imps', 'carb'], ['scheme', 'design', 'ettle', 'nettle', 'sting', 'wiles', 'plans', 'ideas', 'plots', 'cocks'], ['spout', 'nosle', 'snout', 'mouth', 'nostr', 'ports', 'inlet', 'vents', 'outlt', 'beaks'], ['drier', 'arid', 'sere', 'parch', 'dryer', 'wring', 'drear', 'sear', 'pall', 'lack']]

Model response:
To address the issue where the successor state has the same number of filled cells as the parent state, let's revise our approach.

Key Points to Fix:
1. **Incremental Filling**: The successor state should have at least one new cell filled compared to the parent state.
2. **Sequential Filling**: Fill cells row-wise starting with the first incomplete row and column-wise starting with the first incomplete column.
3. **Mixed Priority**: Prioritize horizontal filling first, then vertical.

Steps:
1. **Identify the first empty row for horizontal filling**.
2. **Identify the first empty column for vertical filling**.
3. **Ensure incremental state updates to add at least one new word**.

Revised Successor Function:

```python
def successor_state(state, horizontal_answers, vertical_answers):
    import copy

    def is_valid_state(state):
        for i in range(5):
            # Check each horizontal word
            horizontal_word = ''.join([state[i][j] if state[i][j] is
                not None else '*' for j in range(5)])
            if '*' in horizontal_word:
                fragments = horizontal_word.split('*')
                if fragments[0]:
                    horizontal_word = fragments[0]
                elif len(fragments) > 1:
                    horizontal_word = fragments[1]
                else:
                    horizontal_word = ''
            if horizontal_word and horizontal_word not in
                valid_horizontal_words[i]:
```

```
            return False

        # Check each vertical word
        vertical_word = ''.join([state[j][i] if state[j][i] is not
             None else '*' for j in range(5)])
        if '*' in vertical_word:
            fragments = vertical_word.split('*')
            if fragments[0]:
                vertical_word = fragments[0]
            elif len(fragments) > 1:
                vertical_word = fragments[1]
            else:
                vertical_word = ''
        if vertical_word and vertical_word not in
            valid_vertical_words[i]:
            return False
    return True

def generate_successors(state, possible_answers, is_horizontal):
    successors = []
    for i in range(5):
        if is_horizontal:
            current_line = [state[i][j] for j in range(5)]
        else:
            current_line = [state[j][i] for j in range(5)]

        if all(ch is not None for ch in current_line):  # Skip
            fully filled rows/columns
            continue

        for word in possible_answers[i]:
            if len(word) > 5:
                continue
            new_state = copy.deepcopy(state)
            can_fit = True

            for j in range(5):
                ch = word[j] if j < len(word) else None
                if current_line[j] is not None and current_line[j]
                     != ch:
                    can_fit = False
                    break
                if is_horizontal and ch is not None:
                    new_state[i][j] = ch
                elif not is_horizontal and ch is not None:
                    new_state[j][i] = ch
            if can_fit and is_valid_state(new_state):
                successors.append(new_state)
                break  # Ensure we only get one valid successor at
                     a time
        if successors:
            return successors  # Return successors after
               processing the first row/column with empty cells

    return []

valid_horizontal_words = [set(horizontal_answers[i]) for i in
    range(5)]
valid_vertical_words = [set(vertical_answers[i]) for i in range(5)
    ]

# Try to generate horizontal successors first, if no valid
    horizontal successors, then vertical.
horizontal_successors = generate_successors(state,
    horizontal_answers, True)
```

```
    if horizontal_successors:
        return horizontal_successors

    vertical_successors = generate_successors(state, vertical_answers, False)
    if vertical_successors:
        return vertical_successors

    return []
```

Explanation:
1. **‘is_valid_state(state)‘**: Checks whether partial words formed horizontally and vertically are in the valid set.
   - Handles separating valid parts of the word via ‘’*’‘.
2. **‘generate_successors(state, possible_answers, is_horizontal)‘**:
   - Determines whether to fill rows (horizontal) or columns (vertical).
   - Iterates through each row/column and attempts to place each possible word.
   - It ensures new states have at least one more cell filled by breaking after finding a valid word for the first row/column with ‘None‘ cells.
   - Manages access within bounds (0 to 4) for the 5x5 grid.
3. **Sequential Filling**: Ensures horizontal successors are attempted first. If none, vertical successors are attempted next.

This revised function aims to ensure incremental progression in each successor state, incrementally filling the grid correctly and ensuring valid placements.

Successor Iteration 8

AutoToS prompt:
Successor state has the same number of filled cells as the parent state.
Let’s think step by step. First, think what you did wrong.
Then, think of in what ways successor state should be different from the parent state.
Then, provide the complete Python code for the revised successor function that returns a list of successor states.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: [[None, None, None, None, None], [None, None, None, None, None], [’g’, ’r’, ’a’, ’n’, ’d’], [’v’, ’e’, ’n’, ’u’, ’e’], [’j’, ’e’, ’e’, ’r’, None]]
Example wrong successor state: [[None, None, None, None, None], [None, None, None, None, None], [’g’, ’r’, ’a’, ’n’, ’d’], [’v’, ’e’, ’n’, ’u’, ’e’], [’j’, ’e’, ’e’, ’r’, None]]
Horizontal clues: [[’tasks’, ’goals’, ’plans’, ’agend’, ’chores’, ’works’, ’deeds’, ’items’, ’lists’, ’brief’], [’motor’, ’power’, ’drive’, ’diesel’, ’steam’, ’pumps’, ’crank’, ’gears’, ’turbn’, ’motor’], [’grand’, ’artsy’, ’showy’, ’ornate’, ’fancy’, ’vain’, ’proud’, ’vogue’, ’swank’, ’luxus’], [’venue’, ’salle’, ’forum’, ’atria’, ’lobby’, ’parls’, ’court’, ’malls’, ’mall’, ’lobby’], [’jeer’, ’scoff’, ’sleer’, ’deris’, ’sneer’, ’scorn’, ’derid’, ’gibes’, ’gibed’, ’flout’]]
Vertical clues: [[’amass’, ’stack’, ’hoard’, ’pile’, ’store’, ’heaps’, ’massy’, ’gathe’, ’lumps’, ’mound’], [’nilga’, ’goral’, ’eland’, ’lepus’, ’gazal’, ’kudu’, ’oryx’, ’gnu’, ’imps’, ’carb’], [’scheme’, ’design’, ’ettle’, ’nettle’, ’sting’, ’wiles’, ’plans’, ’ideas’, ’plots’, ’cocks’], [’spout’, ’nosle’, ’snout’, ’mouth’, ’nostr’, ’ports’, ’inlet’, ’vents’, ’outlt’, ’beaks’], [’drier’, ’arid’, ’sere’, ’parch’, ’dryer’, ’wring’, ’drear’, ’sear’, ’pall’, ’lack’]]

Model response:
First, let's analyze why our previous implementations did not generate any successors with a different number of filled cells compared to the parent state:
1. **Mistake Analysis**: The previous logic might be prioritizing certain conditions, hence not covering all possible valid placements. We need to revise how we handle rows and columns to ensure at least one new entry per successor state.
2. **Criteria for Successor State**: A successor should have more filled cells than its parent state and should be selected according to the correct word clues (horizontal or vertical).

Revised Plan
1. **Identify the first row or column with empty cells** and attempt to fill it.
2. **Generate all valid states from the initial state by filling in additional words**.
3. **Each successor state must have at least one additional filled cell** compared to the parent state.
4. **Ensure that word placements do not violate the valid grid state **.

Here's the corrected function:

```python
def successor_state(state, horizontal_answers, vertical_answers):
    import copy

    def is_valid_state(state):
        for i in range(5):
            horizontal_word = ''.join([state[i][j] if state[i][j] is
                not None else '*' for j in range(5)])
            vertical_word = ''.join([state[j][i] if state[j][i] is not
                 None else '*' for j in range(5)])

            if '*' in horizontal_word:
                partials = horizontal_word.split('*')
                if not all(part == '' or part in
                    valid_horizontal_words[i] for part in partials):
                    return False
            elif horizontal_word and horizontal_word not in
                valid_horizontal_words[i]:
                return False

            if '*' in vertical_word:
                partials = vertical_word.split('*')
                if not all(part == '' or part in valid_vertical_words[
                    i] for part in partials):
                    return False
            elif vertical_word and vertical_word not in
                valid_vertical_words[i]:
                return False
        return True

    def generate_successors(state, possible_answers, is_horizontal):
        successors = []
        for i in range(5):
            if is_horizontal:
                current_line = [state[i][j] for j in range(5)]
            else:
                current_line = [state[j][i] for j in range(5)]

            if all(ch is not None for ch in current_line):  # Skip
                fully filled rows/columns
```

```
                continue

            for word in possible_answers[i]:
                if len(word) > 5:
                    continue

                new_state = copy.deepcopy(state)
                can_fit = True

                for j in range(5):
                    ch = word[j] if j < len(word) else None
                    if current_line[j] is not None and current_line[j] != ch:
                        can_fit = False
                        break
                    if is_horizontal and ch is not None:
                        new_state[i][j] = ch
                    elif not is_horizontal and ch is not None:
                        new_state[j][i] = ch

                if can_fit and is_valid_state(new_state):
                    successors.append(new_state)

            if successors:
                return successors  # Stop after generating the first set of valid successors

        return []

    valid_horizontal_words = [set(horizontal_answers[i]) for i in range(5)]
    valid_vertical_words = [set(vertical_answers[i]) for i in range(5)]

    # First try to generate horizontal successors
    row_successors = generate_successors(state, horizontal_answers, True)
    if row_successors:
        return row_successors

    # Then try to generate vertical successors
    col_successors = generate_successors(state, vertical_answers, False)
    if col_successors:
        return col_successors

    return []
```

Explanation:
1. **`is_valid_state(state)`**:
   - Checks for valid partial words both horizontally and vertically.
   - Splits words by `'*'` to manage and check partial phrases.
2. **`generate_successors(state, possible_answers, is_horizontal)`**:
   - Handles both horizontal and vertical filling.
   - Tries placing each word in the specified row/column.
   - Ensures that successors contain at least one additional filled cell compared to the parent state.
   - Only one row or column is processed to ensure incremental state filling.

This approach ensures that each successor correctly extends the state by filling in new words while maintaining validity constraints.

System message:

Successor States Soundness Test Passed
Completeness test
Successor Iteration 9

AutoToS prompt:
Successor function when run on the state [[None, None, 'e', None, None
], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None
, None, 'l', None, None], [None, None, 'e', None, None]] failed to
produce all successors.
Missing successors are: [[['a', 'g', 'e', 'n', 'd'], ['m', 'o', 't', '
o', 'r'], [None, None, 't', None, None], [None, None, 'l', None,
None], [None, None, 'e', None, None]], [['d', 'e', 'e', 'd', 's'],
['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None,
None, 'l', None, None], [None, None, 'e', None, None]], [['i', 't
', 'e', 'm', 's'], ['m', 'o', 't', 'o', 'r'], [None, None, 't',
None, None], [None, None, 'l', None, None], [None, None, 'e', None
, None]], [[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r
'], ['a', 'r', 't', 's', 'y'], [None, None, 'l', None, None], [
None, None, 'e', None, None]], [[None, None, 'e', None, None], ['m
', 'o', 't', 'o', 'r'], [None, None, 't', None, None], ['s', 'a',
'l', 'l', 'e'], [None, None, 'e', None, None]], [[None, None, 'e',
None, None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None,
None], ['m', 'a', 'l', 'l', 's'], [None, None, 'e', None, None]],
[[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None,
None, 't', None, None], [None, None, 'l', None, None], ['s', 'l',
'e', 'e', 'r']], [[None, None, 'e', None, None], ['m', 'o', 't', '
o', 'r'], [None, None, 't', None, None], [None, None, 'l', None,
None], ['s', 'n', 'e', 'e', 'r']], [['a', None, 'e', None, None],
['m', 'o', 't', 'o', 'r'], ['a', None, 't', None, None], ['s',
None, 'l', None, None], ['s', None, 'e', None, None]], [[None, 'g
', 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None, 'r', 't',
None, None], [None, 'a', 'l', None, None], [None, 'l', 'e', None,
None]], [[None, None, 'e', 'n', None], ['m', 'o', 't', 'o', 'r'],
[None, None, 't', 's', None], [None, None, 'l', 'l', None], [None,
None, 'e', 'e', None]], [[None, None, 'e', 'm', None], ['m', 'o',
't', 'o', 'r'], [None, None, 't', 'u', None], [None, None, 'l', '
t', None], [None, None, 'e', 'h', None]], [[None, None, 'e', 'n',
None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', 's', None], [
None, None, 'l', 't', None], [None, None, 'e', 'r', None]], [[None
, None, 'e', 'p', None], ['m', 'o', 't', 'o', 'r'], [None, None, '
t', 'r', None], [None, None, 'l', 't', None], [None, None, 'e', 's
', None]], [[None, None, 'e', None, 'd'], ['m', 'o', 't', 'o', 'r
'], [None, None, 't', None, 'i'], [None, None, 'l', None, 'e'], [
None, None, 'e', None, 'r']], [[None, None, 'e', None, 'd'], ['m',
'o', 't', 'o', 'r'], [None, None, 't', None, 'y'], [None, None, '
l', None, 'e'], [None, None, 'e', None, 'r']], [[None, None, 'e',
None, 'w'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, 'i
'], [None, None, 'l', None, 'n'], [None, None, 'e', None, 'g']],
[[None, None, 'e', None, 'd'], ['m', 'o', 't', 'o', 'r'], [None,
None, 't', None, 'e'], [None, None, 'l', None, 'a'], [None, None,
'e', None, 'r']]]
First think step by step why the successor function failed to produce
all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same
function signature.
Horizontal clues: [['tasks', 'goals', 'plans', 'agend', 'chores', '
works', 'deeds', 'items', 'lists', 'brief'], ['motor', 'power', '
drive', 'diesel', 'steam', 'pumps', 'crank', 'gears', 'turbn', '
motor'], ['grand', 'artsy', 'showy', 'ornate', 'fancy', 'vain', '
proud', 'vogue', 'swank', 'luxus'], ['venue', 'salle', 'forum', '
atria', 'lobby', 'parls', 'court', 'malls', 'mall', 'lobby'], ['
jeer', 'scoff', 'sleer', 'deris', 'sneer', 'scorn', 'derid', '
gibes', 'gibed', 'flout']]

Vertical clues: [['amass', 'stack', 'hoard', 'pile', 'store', 'heaps', 'massy', 'gathe', 'lumps', 'mound'], ['nilga', 'goral', 'eland', 'lepus', 'gazal', 'kudu', 'oryx', 'gnu', 'imps', 'carb'], ['scheme', 'design', 'ettle', 'nettle', 'sting', 'wiles', 'plans', 'ideas', 'plots', 'cocks'], ['spout', 'nosle', 'snout', 'mouth', 'nostr', 'ports', 'inlet', 'vents', 'outlt', 'beaks'], ['drier', 'arid', 'sere', 'parch', 'dryer', 'wring', 'drear', 'sear', 'pall', 'lack']]

Model response:
We need to ensure that the successor function properly fills in additional cells compared to the parent state, and to produce all valid successors. Here's a detailed step-by-step approach to fixing the function:

1. **Reasons for Failure**:
   - The previous function did not sufficiently ensure that new cells were being filled in each successor.
   - It didn't fully cover all possible word placements for both horizontal and vertical fillings.

2. **Successor State Requirements**:
   - Each successor state should introduce at least one new word in an incomplete row or column.
   - Ensure careful iteration over the grid so that all possible valid placements are considered.

Revised Plan:

1. **Identify the incomplete rows and columns**.
2. **Generate all possible successor states for each incomplete row/column**.
3. **Ensure each successor state has at least one more cell filled compared to the parent state**.

Improved Function:

```python
def successor_state(state, horizontal_answers, vertical_answers):
    import copy

    def is_valid_state(state):
        for i in range(5):
            # Check each horizontal word
            horizontal_word = ''.join([state[i][j] if state[i][j] is
                not None else '*' for j in range(5)])
            if '*' in horizontal_word:
                partials = horizontal_word.split('*')
                if not all(part == '' or part in
                    valid_horizontal_words[i] for part in partials):
                    return False
            elif horizontal_word and horizontal_word not in
                valid_horizontal_words[i]:
                return False

            # Check each vertical word
            vertical_word = ''.join([state[j][i] if state[j][i] is not
                 None else '*' for j in range(5)])
            if '*' in vertical_word:
                partials = vertical_word.split('*')
                if not all(part == '' or part in valid_vertical_words[
                    i] for part in partials):
                    return False
            elif vertical_word and vertical_word not in
                valid_vertical_words[i]:
```

```
                return False
        return True

    def generate_successors(state, possible_answers, is_horizontal):
        successors = []
        for i in range(5):
            if is_horizontal:
                target_line = [state[i][j] for j in range(5)]
            else:
                target_line = [state[j][i] for j in range(5)]

            if all(ch is not None for ch in target_line):  # Skip
                fully filled rows/columns
                continue

            for word in possible_answers[i]:
                if len(word) > 5:
                    continue

                new_state = copy.deepcopy(state)
                can_fit = True
                filled = False

                for j in range(len(word)):
                    ch = word[j]
                    if target_line[j] is not None and target_line[j]
                        != ch:
                        can_fit = False
                        break
                    if is_horizontal:
                        new_state[i][j] = ch
                    else:
                        new_state[j][i] = ch
                    filled = filled or (target_line[j] is None and ch
                        is not None)

                if can_fit and filled and is_valid_state(new_state):
                    successors.append(new_state)

            if successors:
                return successors  # Stop after generating the first
                    set of valid successors

        return []

    valid_horizontal_words = [set(horizontal_answers[i]) for i in
        range(5)]
    valid_vertical_words = [set(vertical_answers[i]) for i in range(5)
        ]

    # First try to generate horizontal successors
    row_successors = generate_successors(state, horizontal_answers,
        True)
    if row_successors:
        return row_successors

    # Then try to generate vertical successors
    col_successors = generate_successors(state, vertical_answers,
        False)
    if col_successors:
        return col_successors

    return []
```

```
### Explanation:
1. **`is_valid_state(state)`**: Ensures that partially or fully filled
     rows/columns match the provided clues.
2. **`generate_successors(state, possible_answers, is_horizontal)`**:
   - Iterates through possible words and tries placing each word in
       the specified row/column.
   - Checks if the word matches the pre-filled cells and if at least
       one new cell is filled.
   - Adds successors only if they are valid.
3. **Validation**: Ensures that each newly generated state does not
    conflict with the existing state and adheres to the constraints
    posed by the clues.

This refined function will generate valid successors by ensuring
    incremental updates, each with additional filled cells, covering
    all potential placements for both horizontal and vertical clues.

System message:
Successor States Soundness Test Passed
Completeness test

System message:
Maximum iterations reached for completeness test. Process stopped.
```

9.3.4 Llama3.1 70B

Listing 42: Without partial successor soundness test

```
AutoToS prompt:

The task is a 5x5 mini crosswords. A state is a 2D array representing
    the current puzzle state, where the initial grid is all "None".
    Note that some of the possible answers are not exactly 5 character
     long. Given an input of possible answers to horizontal clues and
    vertical clues, how is the successor state function defined?
    Please first think step by step. Then provide the successor state
    function in Python code.

The possible clues for each row and each column are given separately.
    Here is an example of possible horizontal and vertical clues:

horizontal_answers = [
        ["tasks", "goals", "plans", "agend", "chores", "works", "deeds
            ", "items", "lists", "brief"],
        ["motor", "power", "drive", "diesel", "steam", "pumps", "crank
            ", "gears", "turbn", "motor"],
        ["grand", "artsy", "showy", "ornate", "fancy", "vain", "proud
            ", "vogue", "swank", "luxus"],
        ["venue", "salle", "forum", "atria", "lobby", "parls", "court
            ", "malls", "mall", "lobby"],
        ["jeer", "scoff", "sleer", "deris", "sneer", "scorn", "derid",
             "gibes", "gibed", "flout"]
]
vertical_answers = [
        ["amass", "stack", "hoard", "pile", "store", "heaps", "massy",
             "gathe", "lumps", "mound"],
        ["nilga", "goral", "eland", "lepus", "gazal", "kudu", "oryx",
            "gnu", "imps", "carb"],
        ["scheme", "design", "ettle", "nettle", "sting", "wiles", "
           plans", "ideas", "plots", "cocks"],
        ["spout", "nosle", "snout", "mouth", "nostr", "ports", "inlet
            ", "vents", "outlt", "beaks"],
        ["drier", "arid", "sere", "parch", "dryer", "wring", "drear",
            "sear", "pall", "lack"]
]
```

where horizontal_answers is a list where element i is a list of possible answers to clue in row i, and vertical_answers is a list where element i is a list of possible answers to clue in column i.

Model response:
Defining the Successor State Function
The successor state function generates all possible next states given a current state. In the context of the 5x5 mini crossword puzzle, a state is a 2D array representing the current puzzle state. The successor state function will fill in a single empty cell with a possible answer.
To implement the successor state function, we need to consider the following steps:
Iterate over each empty cell in the current state.
For each empty cell, determine the possible answers based on the horizontal and vertical clues.
For each possible answer, create a new state by filling in the empty cell with the answer.
Python Implementation
Python
```
def successor_state(current_state, horizontal_answers,
    vertical_answers):
    """
    Generates all possible next states given a current state.

    Args:
    current_state (list of lists): A 2D array representing the current
         puzzle state.
    horizontal_answers (list of lists): A list where element i is a
        list of possible answers to clue in row i.
    vertical_answers (list of lists): A list where element i is a list
         of possible answers to clue in column i.

    Returns:
    list of lists of lists: A list of all possible next states.
    """
    next_states = []
    for i in range(len(current_state)):
        for j in range(len(current_state[i])):
            if current_state[i][j] is None:
                # Determine possible answers based on horizontal clue
                if j == 0:  # First column, consider all possible
                    answers
                    possible_answers_horizontal = horizontal_answers[i
                        ]
                else:
                    possible_answers_horizontal = [answer for answer
                        in horizontal_answers[i] if answer[j] ==
                        current_state[i][j - 1]]

                # Determine possible answers based on vertical clue
                if i == 0:  # First row, consider all possible answers
                    possible_answers_vertical = vertical_answers[j]
                else:
                    possible_answers_vertical = [answer for answer in
                        vertical_answers[j] if answer[i] ==
                        current_state[i - 1][j]]

                # Intersection of possible answers from horizontal and
                     vertical clues
                possible_answers = list(set(
                    possible_answers_horizontal) & set(
                    possible_answers_vertical))
```

```
                # Create new states by filling in the empty cell with
                   each possible answer
                for answer in possible_answers:
                    new_state = [row[:] for row in current_state]
                    new_state[i][j] = answer[0]  # Fill in the first
                       character of the answer
                    next_states.append(new_state)

    return next_states
```
Example Usage
Python
```
# Initialize an empty puzzle state
current_state = [[None]*5 for _ in range(5)]

# Define horizontal and vertical clues
horizontal_answers = [
    ["tasks", "goals", "plans", "agend", "chores", "works", "deeds", "
       items", "lists", "brief"],
    ["motor", "power", "drive", "diesel", "steam", "pumps", "crank", "
       gears", "turbn", "motor"],
    ["grand", "artsy", "showy", "ornate", "fancy", "vain", "proud", "
       vogue", "swank", "luxus"],
    ["venue", "salle", "forum", "atria", "lobby", "parls", "court", "
       malls", "mall", "lobby"],
    ["jeer", "scoff", "sleer", "deris", "sneer", "scorn", "derid", "
       gibes", "gibed", "flout"]
]
vertical_answers = [
    ["amass", "stack", "hoard", "pile", "store", "heaps", "massy", "
       gathe", "lumps", "mound"],
    ["nilga", "goral", "eland", "lepus", "gazal", "kudu", "oryx", "gnu
       ", "imps", "carb"],
    ["scheme", "design", "ettle", "nettle", "sting", "wiles", "plans",
        "ideas", "plots", "cocks"],
    ["spout", "nosle", "snout", "mouth", "nostr", "ports", "inlet", "
       vents", "outlt", "beaks"],
    ["drier", "arid", "sere", "parch", "dryer", "wring", "drear", "
       sear", "pall", "lack"]
]

# Generate all possible next states
next_states = successor_state(current_state, horizontal_answers,
   vertical_answers)

# Print the number of next states
print(len(next_states))
```
Note that the number of next states can be very large, depending on the number of possible answers for each clue. This implementation assumes that the puzzle state is a 5x5 grid, but it can be easily extended to support larger or smaller puzzle sizes.
Generated by Meta Llama 3.1-405

AutoToS prompt:

Please provide a goal test function in Python ensuring that all cells are not None, all words horizontally are matching at least one of the horizontal answers, and all words vertically match at least one of the vertical answers. The function should receive a state, horizontal_answers, and vertical_answers as before and return True for a goal state and False otherwise.

Model response:
Goal Test Function

The goal test function checks if a given state is a goal state, i.e., a valid solution to the crossword puzzle. A state is considered a goal state if:

All cells are filled (not None).

All horizontal words match at least one of the horizontal answers.

All vertical words match at least one of the vertical answers.

Python Implementation

Python

```
def is_goal_state(state, horizontal_answers, vertical_answers):
    """
    Checks if a given state is a goal state.

    Args:
    state (list of lists): A 2D array representing the current puzzle state.
    horizontal_answers (list of lists): A list where element i is a list of possible answers to clue in row i.
    vertical_answers (list of lists): A list where element i is a list of possible answers to clue in column i.

    Returns:
    bool: True if the state is a goal state, False otherwise.
    """
    # Check if all cells are filled
    if not all(all(cell is not None for cell in row) for row in state):
        return False

    # Check horizontal words
    for i, row in enumerate(state):
        word = ’’.join(row)
        if not any(word == answer for answer in horizontal_answers[i]):
            return False

    # Check vertical words
    for j in range(len(state[0])):
        word = ’’.join(state[i][j] for i in range(len(state)))
        if not any(word == answer for answer in vertical_answers[j]):
            return False

    # If all checks pass, the state is a goal state
    return True
```

Example Usage

Python

```
# Define a sample state
state = [
    [’t’, ’a’, ’s’, ’k’, ’s’],
    [’m’, ’o’, ’t’, ’o’, ’r’],
    [’g’, ’r’, ’a’, ’n’, ’d’],
    [’v’, ’e’, ’n’, ’u’, ’e’],
    [’j’, ’e’, ’e’, ’r’, ’s’]
]

# Define horizontal and vertical clues
horizontal_answers = [
    ["tasks", "goals", "plans", "agend", "chores", "works", "deeds", "items", "lists", "brief"],
    ["motor", "power", "drive", "diesel", "steam", "pumps", "crank", "gears", "turbn", "motor"],
    ["grand", "artsy", "showy", "ornate", "fancy", "vain", "proud", "vogue", "swank", "luxus"],
```

```
    ["venue", "salle", "forum", "atria", "lobby", "parls", "court", "
        malls", "mall", "lobby"],
    ["jeer", "scoff", "sleer", "deris", "sneer", "scorn", "derid", "
        gibes", "gibed", "flout"]
]
vertical_answers = [
    ["amass", "stack", "hoard", "pile", "store", "heaps", "massy", "
        gathe", "lumps", "mound"],
    ["nilga", "goral", "eland", "lepus", "gazal", "kudu", "oryx", "gnu
        ", "imps", "carb"],
    ["scheme", "design", "ettle", "nettle", "sting", "wiles", "plans",
         "ideas", "plots", "cocks"],
    ["spout", "nosle", "snout", "mouth", "nostr", "ports", "inlet", "
        vents", "outlt", "beaks"],
    ["drier", "arid", "sere", "parch", "dryer", "wring", "drear", "
        sear", "pall", "lack"]
]

# Check if the state is a goal state
if is_goal_state(state, horizontal_answers, vertical_answers):
    print("The state is a goal state.")
else:
    print("The state is not a goal state.")
```

In this example, the is_goal_state function checks if the provided state is a valid solution to the crossword puzzle based on the given horizontal and vertical answers.
Generated by Meta Llama 3.1-405

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
The following exception was produced when testing the successor function:
  File "<string>", line 23, in successor_state

string index out of range. Please fix this exception in the successor function.
Model response:
The "string index out of range" exception occurs when you try to access an index in a string that does not exist. In the context of the successor_state function, this error is likely happening when checking if a possible answer matches the existing characters in the row or column.
To fix this issue, we need to ensure that we're not trying to access an index that is out of range. Here's the corrected successor_state function:
Python

```
def successor_state(current_state, horizontal_answers,
    vertical_answers):
    """
    Generates all possible next states given a current state.

    Args:
    current_state (list of lists): A 2D array representing the current
         puzzle state.
    horizontal_answers (list of lists): A list where element i is a
        list of possible answers to clue in row i.
    vertical_answers (list of lists): A list where element i is a list
         of possible answers to clue in column i.

    Returns:
```

```
    list of lists of lists: A list of all possible next states.
    """
    next_states = []
    for i in range(len(current_state)):
        for j in range(len(current_state[i])):
            if current_state[i][j] is None:
                # Determine possible answers based on horizontal clue
                possible_answers_horizontal = []
                for answer in horizontal_answers[i]:
                    if j == 0 or (j < len(answer) and answer[j] ==
                        current_state[i][j - 1]):
                        possible_answers_horizontal.append(answer)

                # Determine possible answers based on vertical clue
                possible_answers_vertical = []
                for answer in vertical_answers[j]:
                    if i == 0 or (i < len(answer) and answer[i] ==
                        current_state[i - 1][j]):
                        possible_answers_vertical.append(answer)

                # Intersection of possible answers from horizontal and
                     vertical clues
                possible_answers = list(set(
                    possible_answers_horizontal) & set(
                    possible_answers_vertical))

                # Create new states by filling in the empty cell with
                    each possible answer
                for answer in possible_answers:
                    if j < len(answer):  # Check if the answer is long
                         enough
                        new_state = [row[:] for row in current_state]
                        new_state[i][j] = answer[j]  # Fill in the
                            correct character of the answer
                        next_states.append(new_state)

    return next_states
```
In this corrected version, I've added checks to ensure that we're not trying to access an index that is out of range when checking if a possible answer matches the existing characters in the row or column. I've also added a check to ensure that the answer is long enough before trying to access its characters.
Generated by Meta Llama 3.1-405

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 2

AutoToS prompt:
Successor function when run on the state [[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None, None, 'l', None, None], [None, None, 'e', None, None]] failed to produce all successors.
Missing successors are: [[['a', 'g', 'e', 'n', 'd'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None, None, 'l', None, None], [None, None, 'e', None, None]], [['d', 'e', 'e', 'd', 's'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None, None, 'l', None, None], [None, None, 'e', None, None]], [['i', 't', 'e', 'm', 's'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None, None, 'l', None, None], [None, None, 'e', None, None]], [[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], ['a', 'r', 't', 's', 'y'], [None, None, 'l', None, None], [None, None, 'e', None, None]], [[None, None, 'e', None, None], ['m

', 'o', 't', 'o', 'r'], [None, None, 't', None, None], ['s', 'a', 'l', 'l', 'e'], [None, None, 'e', None, None]], [[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], ['m', 'a', 'l', 'l', 's'], [None, None, 'e', None, None]], [[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None, None, 'l', None, None], ['s', 'l', 'e', 'e', 'r']], [[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None, None, 'l', None, None], ['s', 'n', 'e', 'e', 'r']], [['a', None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], ['a', None, 't', None, None], ['s', None, 'l', None, None], ['s', None, 'e', None, None]], [[None, 'g', 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None, 'r', 't', None, None], [None, 'a', 'l', None, None], [None, 'l', 'e', None, None]], [[None, None, 'e', 'n', None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', 's', None], [None, None, 'l', 'l', None], [None, 'e', 'e', None]], [[None, None, 'e', 'm', None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', 'u', None], [None, None, 'l', 't', None], [None, None, 'e', 'h', None]], [[None, None, 'e', 'n', None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', 's', None], [None, None, 'l', 't', None], [None, None, 'e', 'r', None]], [[None, None, 'e', 'p', None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', 'r', None], [None, None, 'l', 't', None], [None, None, 'e', 's', None]], [[None, None, 'e', None, 'd'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, 'i'], [None, None, 'l', None, 'e'], [None, None, 'e', None, 'r']], [[None, None, 'e', None, 'd'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, 'y'], [None, None, 'l', None, 'e'], [None, None, 'e', None, 'r']], [[None, None, 'e', None, 'w'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, 'i'], [None, None, 'l', None, 'n'], [None, None, 'e', None, 'g']], [[None, None, 'e', None, 'd'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, 'e'], [None, None, 'l', None, 'a'], [None, None, 'e', None, 'r']]]

First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.
Horizontal clues: [['tasks', 'goals', 'plans', 'agend', 'chores', 'works', 'deeds', 'items', 'lists', 'brief'], ['motor', 'power', 'drive', 'diesel', 'steam', 'pumps', 'crank', 'gears', 'turbn', 'motor'], ['grand', 'artsy', 'showy', 'ornate', 'fancy', 'vain', 'proud', 'vogue', 'swank', 'luxus'], ['venue', 'salle', 'forum', 'atria', 'lobby', 'parls', 'court', 'malls', 'mall', 'lobby'], ['jeer', 'scoff', 'sleer', 'deris', 'sneer', 'scorn', 'derid', 'gibes', 'gibed', 'flout']]
Vertical clues: [['amass', 'stack', 'hoard', 'pile', 'store', 'heaps', 'massy', 'gathe', 'lumps', 'mound'], ['nilga', 'goral', 'eland', 'lepus', 'gazal', 'kudu', 'oryx', 'gnu', 'imps', 'carb'], ['scheme', 'design', 'ettle', 'nettle', 'sting', 'wiles', 'plans', 'ideas', 'plots', 'cocks'], ['spout', 'nosle', 'snout', 'mouth', 'nostr', 'ports', 'inlet', 'vents', 'outlt', 'beaks'], ['drier', 'arid', 'sere', 'parch', 'dryer', 'wring', 'drear', 'sear', 'pall', 'lack']]

Model response:
The successor function failed to produce all successors of the state because it only considered filling in the empty cell with the first character of each possible answer. However, it should consider filling in the empty cell with any character from the possible answers that match the existing characters in the row or column.
Here's the corrected successor_state function:
Python
```
```

```
def successor_state(current_state, horizontal_answers,
    vertical_answers):
    """
    Generates all possible next states given a current state.

    Args:
    current_state (list of lists): A 2D array representing the current
         puzzle state.
    horizontal_answers (list of lists): A list where element i is a
        list of possible answers to clue in row i.
    vertical_answers (list of lists): A list where element i is a list
         of possible answers to clue in column i.

    Returns:
    list of lists of lists: A list of all possible next states.
    """
    next_states = []
    for i in range(len(current_state)):
        for j in range(len(current_state[i])):
            if current_state[i][j] is None:
                # Determine possible answers based on horizontal clue
                possible_answers_horizontal = []
                for answer in horizontal_answers[i]:
                    if j < len(answer) and (j == 0 or answer[j - 1] ==
                         current_state[i][j - 1]):
                        possible_answers_horizontal.append(answer)

                # Determine possible answers based on vertical clue
                possible_answers_vertical = []
                for answer in vertical_answers[j]:
                    if i < len(answer) and (i == 0 or answer[i - 1] ==
                         current_state[i - 1][j]):
                        possible_answers_vertical.append(answer)

                # Intersection of possible answers from horizontal and
                     vertical clues
                possible_answers = list(set(
                    possible_answers_horizontal) & set(
                    possible_answers_vertical))

                # Create new states by filling in the empty cell with
                    each possible answer
                for answer in possible_answers:
                    if j < len(answer):  # Check if the answer is long
                         enough
                        new_state = [row[:] for row in current_state]
                        new_state[i][j] = answer[j]  # Fill in the
                            correct character of the answer
                        next_states.append(new_state)

                # Create new states by filling in the empty cell with
                    each possible character from the possible answers
                for answer in possible_answers_horizontal:
                    if j < len(answer):  # Check if the answer is long
                         enough
                        new_state = [row[:] for row in current_state]
                        new_state[i][j] = answer[j]  # Fill in the
                            correct character of the answer
                        next_states.append(new_state)

                for answer in possible_answers_vertical:
                    if i < len(answer):  # Check if the answer is long
                         enough
                        new_state = [row[:] for row in current_state]
```

```
                    new_state[i][j] = answer[i]  # Fill in the
                        correct character of the answer
                    next_states.append(new_state)

    return next_states
```
In this corrected version, we're considering filling in the empty cell with any character from the possible answers that match the existing characters in the row or column. We're also creating new states by filling in the empty cell with each possible character from the possible answers.

However, please note that this approach might produce duplicate states, which could be inefficient for large puzzles. A more efficient approach would be to keep track of the possible answers for each cell and update them as we fill in the cells. This way, we can avoid generating duplicate states and reduce the search space.

Generated by Meta Llama 3.1-405

Successor Iteration 3

AutoToS prompt:

Timeout occurred during search with the successor function.

Please provide a different implementation of the successor function.

Keep the same function signature.

Horizontal clues: [['tasks', 'goals', 'plans', 'agend', 'chores', 'works', 'deeds', 'items', 'lists', 'brief'], ['motor', 'power', 'drive', 'diesel', 'steam', 'pumps', 'crank', 'gears', 'turbn', 'motor'], ['grand', 'artsy', 'showy', 'ornate', 'fancy', 'vain', 'proud', 'vogue', 'swank', 'luxus'], ['venue', 'salle', 'forum', 'atria', 'lobby', 'parls', 'court', 'malls', 'mall', 'lobby'], ['jeer', 'scoff', 'sleer', 'deris', 'sneer', 'scorn', 'derid', 'gibes', 'gibed', 'flout']]

Vertical clues: [['amass', 'stack', 'hoard', 'pile', 'store', 'heaps', 'massy', 'gathe', 'lumps', 'mound'], ['nilga', 'goral', 'eland', 'lepus', 'gazal', 'kudu', 'oryx', 'gnu', 'imps', 'carb'], ['scheme', 'design', 'ettle', 'nettle', 'sting', 'wiles', 'plans', 'ideas', 'plots', 'cocks'], ['spout', 'nosle', 'snout', 'mouth', 'nostr', 'ports', 'inlet', 'vents', 'outlt', 'beaks'], ['drier', 'arid', 'sere', 'parch', 'dryer', 'wring', 'drear', 'sear', 'pall', 'lack']]

Model response:

To improve the efficiency of the successor function and avoid timeouts, we can try to reduce the number of possible next states by only considering the answers that match the existing characters in the row or column. Here's an alternative implementation of the successor_state function:

Python
```
def successor_state(current_state, horizontal_answers,
    vertical_answers):
    """
    Generates all possible next states given a current state.

    Args:
    current_state (list of lists): A 2D array representing the current
         puzzle state.
    horizontal_answers (list of lists): A list where element i is a
        list of possible answers to clue in row i.
    vertical_answers (list of lists): A list where element i is a list
         of possible answers to clue in column i.

    Returns:
    list of lists of lists: A list of all possible next states.
    """
    next_states = []
```

```
    for i in range(len(current_state)):
        for j in range(len(current_state[i])):
            if current_state[i][j] is None:
                # Determine possible answers based on horizontal clue
                possible_answers_horizontal = [answer for answer in
                    horizontal_answers[i] if len(answer) > j and all(
                    answer[k] == current_state[i][k] for k in range(j)
                     if current_state[i][k] is not None)]

                # Determine possible answers based on vertical clue
                possible_answers_vertical = [answer for answer in
                    vertical_answers[j] if len(answer) > i and all(
                    answer[k] == current_state[k][j] for k in range(i)
                     if current_state[k][j] is not None)]

                # Intersection of possible answers from horizontal and
                     vertical clues
                possible_answers = list(set(
                    possible_answers_horizontal) & set(
                    possible_answers_vertical))

                # Create new states by filling in the empty cell with
                    each possible answer
                for answer in possible_answers:
                    new_state = [row[:] for row in current_state]
                    new_state[i][j] = answer[j]  # Fill in the correct
                         character of the answer
                    next_states.append(new_state)

    return next_states
```

In this version, we're filtering the possible answers based on the existing characters in the row or column, which should reduce the number of possible next states and improve the efficiency of the search.
Additionally, we can consider using a more efficient data structure, such as a trie or a prefix tree, to store the possible answers and quickly retrieve the matching answers.
If the search is still too slow, we can consider using a more advanced search algorithm, such as A* search or iterative deepening depth-first search, which can take advantage of heuristics to guide the search towards the most promising areas of the search space.
Generated by Meta Llama 3.1-405

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 4

AutoToS prompt:
Successor function when run on the state [[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None, None, 'l', None, None], [None, None, 'e', None, None]] failed to produce all successors.
Missing successors are: [[['a', 'g', 'e', 'n', 'd'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None, None, 'l', None, None], [None, None, 'e', None, None]], [['d', 'e', 'e', 'd', 's'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None, None, 'l', None, None], [None, None, 'e', None, None]], [['i', 't', 'e', 'm', 's'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None, None, 'l', None, None], [None, None, 'e', None, None]], [[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], ['a', 'r', 't', 's', 'y'], [None, None, 'l', None, None], [None, None, 'e', None, None]], [[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], ['s', 'a',

```
    'l', 'l', 'e'], [None, None, 'e', None, None]], [[None, None, 'e',
     None, None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None,
    None], ['m', 'a', 'l', 'l', 's'], [None, None, 'e', None, None]],
    [[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None,
    None, 't', None, None], [None, None, 'l', None, None], ['s', 'l',
    'e', 'e', 'r']], [[None, None, 'e', None, None], ['m', 'o', 't', '
    o', 'r'], [None, None, 't', None, None], [None, None, 'l', None,
    None], ['s', 'n', 'e', 'e', 'r']], [['a', None, 'e', None, None],
    ['m', 'o', 't', 'o', 'r'], ['a', None, 't', None, None], ['s',
    None, 'l', None, None], ['s', None, 'e', None, None]], [[None, 'g
    ', 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None, 'r', 't',
    None, None], [None, 'a', 'l', None, None], [None, 'l', 'e', None,
    None]], [[None, None, 'e', 'n', None], ['m', 'o', 't', 'o', 'r'],
    [None, None, 't', 's', None], [None, None, 'l', 'l', None], [None,
     None, 'e', 'e', None]], [[None, None, 'e', 'm', None], ['m', 'o',
     't', 'o', 'r'], [None, None, 't', 'u', None], [None, None, 'l', '
    t', None], [None, None, 'e', 'h', None]], [[None, None, 'e', 'n',
    None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', 's', None], [
    None, None, 'l', 't', None], [None, None, 'e', 'r', None]], [[None
    , None, 'e', 'p', None], ['m', 'o', 't', 'o', 'r'], [None, None, '
    t', 'r', None], [None, None, 'l', 't', None], [None, None, 'e', 's
    ', None]], [[None, None, 'e', None, 'd'], ['m', 'o', 't', 'o', 'r
    '], [None, None, 't', None, 'i'], [None, None, 'l', None, 'e'], [
    None, None, 'e', None, 'r']], [[None, None, 'e', None, 'd'], ['m',
     'o', 't', 'o', 'r'], [None, None, 't', None, 'y'], [None, None, '
    l', None, 'e'], [None, None, 'e', None, 'r']], [[None, None, 'e',
    None, 'w'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, 'i
    '], [None, None, 'l', None, 'n'], [None, None, 'e', None, 'g']],
    [[None, None, 'e', None, 'd'], ['m', 'o', 't', 'o', 'r'], [None,
    None, 't', None, 'e'], [None, None, 'l', None, 'a'], [None, None,
    'e', None, 'r']]]
First think step by step why the successor function failed to produce
    all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.
Horizontal clues: [['tasks', 'goals', 'plans', 'agend', 'chores', '
    works', 'deeds', 'items', 'lists', 'brief'], ['motor', 'power', '
    drive', 'diesel', 'steam', 'pumps', 'crank', 'gears', 'turbn', '
    motor'], ['grand', 'artsy', 'showy', 'ornate', 'fancy', 'vain', '
    proud', 'vogue', 'swank', 'luxus'], ['venue', 'salle', 'forum', '
    atria', 'lobby', 'parls', 'court', 'malls', 'mall', 'lobby'], ['
    jeer', 'scoff', 'sleer', 'deris', 'sneer', 'scorn', 'derid', '
    gibes', 'gibed', 'flout']]
Vertical clues: [['amass', 'stack', 'hoard', 'pile', 'store', 'heaps',
     'massy', 'gathe', 'lumps', 'mound'], ['nilga', 'goral', 'eland',
    'lepus', 'gazal', 'kudu', 'oryx', 'gnu', 'imps', 'carb'], ['scheme
    ', 'design', 'ettle', 'nettle', 'sting', 'wiles', 'plans', 'ideas
    ', 'plots', 'cocks'], ['spout', 'nosle', 'snout', 'mouth', 'nostr
    ', 'ports', 'inlet', 'vents', 'outlt', 'beaks'], ['drier', 'arid',
     'sere', 'parch', 'dryer', 'wring', 'drear', 'sear', 'pall', 'lack
    ']]

Model response:
The successor function failed to produce all successors of the state
    because it only considered filling in the empty cell with the
    first character of each possible answer. However, it should
    consider filling in the empty cell with any character from the
    possible answers that match the existing characters in the row or
    column.
To fix this issue, we need to modify the successor function to
    consider all possible characters from the possible answers. Here's
     the corrected successor_state function:
Python
```

```
def successor_state(current_state, horizontal_answers,
    vertical_answers):
    """
    Generates all possible next states given a current state.

    Args:
    current_state (list of lists): A 2D array representing the current
         puzzle state.
    horizontal_answers (list of lists): A list where element i is a
        list of possible answers to clue in row i.
    vertical_answers (list of lists): A list where element i is a list
         of possible answers to clue in column i.

    Returns:
    list of lists of lists: A list of all possible next states.
    """
    next_states = []
    for i in range(len(current_state)):
        for j in range(len(current_state[i])):
            if current_state[i][j] is None:
                # Determine possible answers based on horizontal clue
                possible_answers_horizontal = [answer for answer in
                    horizontal_answers[i] if len(answer) > j and all(
                    answer[k] == current_state[i][k] for k in range(j)
                     if current_state[i][k] is not None)]

                # Determine possible answers based on vertical clue
                possible_answers_vertical = [answer for answer in
                    vertical_answers[j] if len(answer) > i and all(
                    answer[k] == current_state[k][j] for k in range(i)
                     if current_state[k][j] is not None)]

                # Create new states by filling in the empty cell with
                    each possible character from the possible answers
                for answer in possible_answers_horizontal:
                    new_state = [row[:] for row in current_state]
                    new_state[i][j] = answer[j]  # Fill in the correct
                         character of the answer
                    next_states.append(new_state)

                for answer in possible_answers_vertical:
                    new_state = [row[:] for row in current_state]
                    new_state[i][j] = answer[i]  # Fill in the correct
                         character of the answer
                    next_states.append(new_state)

                # Create new states by filling in the entire row or
                    column with each possible answer
                for answer in possible_answers_horizontal:
                    new_state = [row[:] for row in current_state]
                    for k in range(len(answer)):
                        new_state[i][k] = answer[k]  # Fill in the
                            entire row with the answer
                    next_states.append(new_state)

                for answer in possible_answers_vertical:
                    new_state = [row[:] for row in current_state]
                    for k in range(len(answer)):
                        new_state[k][j] = answer[k]  # Fill in the
                            entire column with the answer
                    next_states.append(new_state)

    return next_states
```

In this corrected version, we're considering filling in the empty cell with any character from the possible answers that match the existing characters in the row or column. We're also creating new states by filling in the entire row or column with each possible answer.
However, please note that this approach might produce duplicate states, which could be inefficient for large puzzles. A more efficient approach would be to keep track of the possible answers for each cell and update them as we fill in the cells. This way, we can avoid generating duplicate states and reduce the search space.
Generated by Meta Llama 3.1-405
Successor Iteration 5

AutoToS prompt:
The following exception was produced when testing the successor function:
  File "<string>", line 40, in successor_state

list assignment index out of range. Please fix this exception in the successor function.
Model response:
The "list assignment index out of range" exception occurs when you try to assign a value to an index in a list that does not exist. In the context of the successor_state function, this error is likely happening when trying to fill in the entire row or column with each possible answer.
To fix this issue, we need to ensure that we're not trying to assign a value to an index that is out of range. Here's the corrected successor_state function:
Python
```
def successor_state(current_state, horizontal_answers,
    vertical_answers):
    """
    Generates all possible next states given a current state.

    Args:
    current_state (list of lists): A 2D array representing the current
         puzzle state.
    horizontal_answers (list of lists): A list where element i is a
        list of possible answers to clue in row i.
    vertical_answers (list of lists): A list where element i is a list
         of possible answers to clue in column i.

    Returns:
    list of lists of lists: A list of all possible next states.
    """
    next_states = []
    for i in range(len(current_state)):
        for j in range(len(current_state[i])):
            if current_state[i][j] is None:
                # Determine possible answers based on horizontal clue
                possible_answers_horizontal = [answer for answer in
                    horizontal_answers[i] if len(answer) > j and all(
                    answer[k] == current_state[i][k] for k in range(j)
                     if current_state[i][k] is not None)]

                # Determine possible answers based on vertical clue
                possible_answers_vertical = [answer for answer in
                    vertical_answers[j] if len(answer) > i and all(
                    answer[k] == current_state[k][j] for k in range(i)
                     if current_state[k][j] is not None)]

                # Create new states by filling in the empty cell with
                    each possible character from the possible answers
```

```
                for answer in possible_answers_horizontal:
                    new_state = [row[:] for row in current_state]
                    new_state[i][j] = answer[j]  # Fill in the correct
                         character of the answer
                    next_states.append(new_state)

                for answer in possible_answers_vertical:
                    new_state = [row[:] for row in current_state]
                    new_state[i][j] = answer[i]  # Fill in the correct
                         character of the answer
                    next_states.append(new_state)

                # Create new states by filling in the entire row or
                    column with each possible answer
                for answer in possible_answers_horizontal:
                    new_state = [row[:] for row in current_state]
                    for k in range(min(len(answer), len(current_state[
                        i]))):
                        new_state[i][k] = answer[k]  # Fill in the
                            entire row with the answer
                    next_states.append(new_state)

                for answer in possible_answers_vertical:
                    new_state = [row[:] for row in current_state]
                    for k in range(min(len(answer), len(current_state)
                        )):
                        new_state[k][j] = answer[k]  # Fill in the
                            entire column with the answer
                    next_states.append(new_state)

    return next_states
```

```
In this corrected version, I've added checks to ensure that we're not
    trying to assign a value to an index that is out of range when
    filling in the entire row or column with each possible answer. I'
    ve used the min function to ensure that we don't try to access an
    index that is out of range.
However, please note that this approach might still produce duplicate
    states, which could be inefficient for large puzzles. A more
    efficient approach would be to keep track of the possible answers
    for each cell and update them as we fill in the cells. This way,
    we can avoid generating duplicate states and reduce the search
    space.
Generated by Meta Llama 3.1-405

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 20, failure: 0, total expansions 27137742, total generated
    118405140
```

Listing 43: With partial successor soundness test

```
AutoToS prompt:

The task is a 5x5 mini crosswords. A state is a 2D array representing
    the current puzzle state, where the initial grid is all "None".
    Note that some of the possible answers are not exactly 5 character
     long. Given an input of possible answers to horizontal clues and
    vertical clues, how is the successor state function defined?
    Please first think step by step. Then provide the successor state
    function in Python code.
```

The possible clues for each row and each column are given separately. Here is an example of possible horizontal and vertical clues:

```
horizontal_answers = [
        ["tasks", "goals", "plans", "agend", "chores", "works", "deeds
            ", "items", "lists", "brief"],
        ["motor", "power", "drive", "diesel", "steam", "pumps", "crank
            ", "gears", "turbn", "motor"],
        ["grand", "artsy", "showy", "ornate", "fancy", "vain", "proud
            ", "vogue", "swank", "luxus"],
        ["venue", "salle", "forum", "atria", "lobby", "parls", "court
            ", "malls", "mall", "lobby"],
        ["jeer", "scoff", "sleer", "deris", "sneer", "scorn", "derid",
             "gibes", "gibed", "flout"]
]
vertical_answers = [
        ["amass", "stack", "hoard", "pile", "store", "heaps", "massy",
             "gathe", "lumps", "mound"],
        ["nilga", "goral", "eland", "lepus", "gazal", "kudu", "oryx",
            "gnu", "imps", "carb"],
        ["scheme", "design", "ettle", "nettle", "sting", "wiles", "
            plans", "ideas", "plots", "cocks"],
        ["spout", "nosle", "snout", "mouth", "nostr", "ports", "inlet
            ", "vents", "outlt", "beaks"],
        ["drier", "arid", "sere", "parch", "dryer", "wring", "drear",
            "sear", "pall", "lack"]
]
```

where horizontal_answers is a list where element i is a list of possible answers to clue in row i, and vertical_answers is a list where element i is a list of possible answers to clue in column i.

Model response:
Here is the successor state function in Python code:
Python

```
def successor_state(state, horizontal_answers, vertical_answers):
    def is_valid_word(word, row, col):
        if len(word) != 5:
            return False
        for i in range(5):
            if state[row][col + i] is not None and state[row][col + i]
                 != word[i]:
                return False
        return True

    def is_valid_placement(word, row, col):
        for i in range(5):
            if col + i >= 5:
                return False
            if state[row][col + i] is not None and state[row][col + i]
                 != word[i]:
                return False
        return True

    def place_word(word, row, col):
        new_state = [r[:] for r in state]
        for i in range(5):
            new_state[row][col + i] = word[i]
        return new_state

    successors = []
    for row in range(5):
        for col in range(5):
            if state[row][col] is None:
```

```
                for word in horizontal_answers[row]:
                    if is_valid_placement(word, row, col):
                        successors.append(place_word(word, row, col))
                break
        else:
            for col in range(5):
                for word in vertical_answers[col]:
                    if is_valid_word(word, row, col):
                        successors.append(place_word(word, row, col))
    return successors
```

This function generates all possible successor states by trying to place each possible word in each empty space in the grid. It checks if the word can be placed horizontally or vertically without conflicting with existing letters. If a word can be placed , it creates a new state with the word placed and adds it to the list of successors

AutoToS prompt:

Please provide a goal test function in Python ensuring that all cells are not None, all words horizontally are matching at least one of the horizontal answers, and all words vertically match at least one of the vertical answers. The function should receive a state, horizontal_answers, and vertical_answers as before and return True for a goal state and False otherwise.

Model response:
Here is the goal test function in Python:
Python

```
def goal_test(state, horizontal_answers, vertical_answers):
    def is_valid_word(word, answers):
        return word in answers

    # Check all cells are not None
    for row in state:
        for cell in row:
            if cell is None:
                return False

    # Check all horizontal words match at least one answer
    for row in range(5):
        word = ''.join(state[row])
        if not is_valid_word(word, horizontal_answers[row]):
            return False

    # Check all vertical words match at least one answer
    for col in range(5):
        word = ''.join([state[row][col] for row in range(5)])
        if not is_valid_word(word, vertical_answers[col]):
            return False

    return True
```

This function first checks if all cells in the state are not None. Then it checks if each horizontal word in the state matches at least one of the possible horizontal answers. Finally, it checks if each vertical word in the state matches at least one of the possible vertical answers. If all these conditions are met, it returns True, indicating a goal state; otherwise, it returns False

System message:
Goal Soundness Test Passed
Soundness test

Successor Iteration 1

AutoToS prompt:
The following exception was produced when testing the successor function:
  File "<string>", line 23, in place_word

string index out of range. Please fix this exception in the successor function.
Model response:
The exception "string index out of range" occurs when trying to access an index in a string that doesn’t exist. In this case, it’s happening in the place_word function when trying to access word[i].
This error occurs because the place_word function assumes that the word has at least 5 characters, but some words in the horizontal_answers and vertical_answers lists have fewer than 5 characters.
Here’s the corrected successor_state function:
Python
```
def successor_state(state, horizontal_answers, vertical_answers):
    def is_valid_word(word, row, col):
        if len(word) != 5:
            return False
        for i in range(5):
            if state[row][col + i] is not None and state[row][col + i] != word[i]:
                return False
        return True

    def is_valid_placement(word, row, col):
        if col + len(word) > 5:
            return False
        for i in range(len(word)):
            if col + i >= 5 or (state[row][col + i] is not None and state[row][col + i] != word[i]):
                return False
        return True

    def place_word(word, row, col):
        new_state = [r[:] for r in state]
        for i in range(len(word)):
            new_state[row][col + i] = word[i]
        return new_state

    successors = []
    for row in range(5):
        for col in range(5):
            if state[row][col] is None:
                for word in horizontal_answers[row]:
                    if is_valid_placement(word, row, col):
                        successors.append(place_word(word, row, col))
                break
        else:
            for col in range(5):
                for word in vertical_answers[col]:
                    if is_valid_word(word, row, col):
                        successors.append(place_word(word, row, col))
    return successors
```
In the corrected version, I’ve changed the is_valid_placement function to check if the word fits in the remaining space in the row, and I’ve also changed the place_word function to iterate over the length of the word instead of assuming it’s always 5 characters

Successor Iteration 2

AutoToS prompt:
The following exception was produced when testing the successor function:
  File "<string>", line 8, in is_valid_word

list index out of range. Please fix this exception in the successor function.
Model response:
The exception "list index out of range" occurs when trying to access an index in a list that doesn’t exist. In this case, it’s happening in the is_valid_word function when trying to access state[row][col + i].
This error occurs because the is_valid_word function assumes that the row has at least col + 5 columns, but some rows have fewer columns .
Here’s the corrected successor_state function:
Python
```
def successor_state(state, horizontal_answers, vertical_answers):
    def is_valid_word(word, row, col):
        if col + len(word) > 5:
            return False
        for i in range(len(word)):
            if col + i >= 5 or (state[row][col + i] is not None and
                state[row][col + i] != word[i]):
                return False
        return True

    def is_valid_placement(word, row, col):
        if col + len(word) > 5:
            return False
        for i in range(len(word)):
            if col + i >= 5 or (state[row][col + i] is not None and
                state[row][col + i] != word[i]):
                return False
        return True

    def place_word(word, row, col):
        new_state = [r[:] for r in state]
        for i in range(len(word)):
            new_state[row][col + i] = word[i]
        return new_state

    successors = []
    for row in range(5):
        for col in range(5):
            if state[row][col] is None:
                for word in horizontal_answers[row]:
                    if is_valid_placement(word, row, col):
                        successors.append(place_word(word, row, col))
                break
        else:
            for col in range(5):
                for word in vertical_answers[col]:
                    if is_valid_word(word, row, col):
                        successors.append(place_word(word, row, col))
    return successors
```
In the corrected version, I’ve added a check in the is_valid_word function to ensure that col + len(word) doesn’t exceed the number of columns in the row. This prevents the "list index out of range" error

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 3

AutoToS prompt:
Successor function when run on the state [[None, None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, None], [None, None, 'l', None, None], [None, None, 'e', None, None]] failed to produce all successors.
Missing successors are: [[['a', None, 'e', None, None], ['m', 'o', 't', 'o', 'r'], ['a', None, 't', None, None], ['s', None, 'l', None, None], ['s', None, 'e', None, None]], [[None, 'g', 'e', None, None], ['m', 'o', 't', 'o', 'r'], [None, 'r', 't', None, None], [None, 'a', 'l', None, None], [None, 'l', 'e', None, None]], [[None, None, 'e', 'n', None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', 's', None], [None, None, 'l', 'l', None], [None, None, 'e', 'e', None]], [[None, None, 'e', 'm', None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', 'u', None], [None, None, 'l', 't', None], [None, None, 'e', 'h', None]], [[None, None, 'e', 'n', None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', 's', None], [None, None, 'l', 't', None], [None, None, 'e', 'r', None]], [[None, None, 'e', 'p', None], ['m', 'o', 't', 'o', 'r'], [None, None, 't', 'r', None], [None, None, 'l', 't', None], [None, None, 'e', 's', None]], [[None, None, 'e', None, 'd'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, 'i'], [None, None, 'l', None, 'e'], [None, None, 'e', None, 'r']], [[None, None, 'e', None, 'd'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, 'y'], [None, None, 'l', None, 'e'], [None, None, 'e', None, 'r']], [[None, None, 'e', None, 'w'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, 'i'], [None, None, 'l', None, 'n'], [None, None, 'e', None, 'g']], [[None, None, 'e', None, 'd'], ['m', 'o', 't', 'o', 'r'], [None, None, 't', None, 'e'], [None, None, 'l', None, 'a'], [None, None, 'e', None, 'r']]]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.
Horizontal clues: [['tasks', 'goals', 'plans', 'agend', 'chores', 'works', 'deeds', 'items', 'lists', 'brief'], ['motor', 'power', 'drive', 'diesel', 'steam', 'pumps', 'crank', 'gears', 'turbn', 'motor'], ['grand', 'artsy', 'showy', 'ornate', 'fancy', 'vain', 'proud', 'vogue', 'swank', 'luxus'], ['venue', 'salle', 'forum', 'atria', 'lobby', 'parls', 'court', 'malls', 'mall', 'lobby'], ['jeer', 'scoff', 'sleer', 'deris', 'sneer', 'scorn', 'derid', 'gibes', 'gibed', 'flout']]
Vertical clues: [['amass', 'stack', 'hoard', 'pile', 'store', 'heaps', 'massy', 'gathe', 'lumps', 'mound'], ['nilga', 'goral', 'eland', 'lepus', 'gazal', 'kudu', 'oryx', 'gnu', 'imps', 'carb'], ['scheme', 'design', 'ettle', 'nettle', 'sting', 'wiles', 'plans', 'ideas', 'plots', 'cocks'], ['spout', 'nosle', 'snout', 'mouth', 'nostr', 'ports', 'inlet', 'vents', 'outlt', 'beaks'], ['drier', 'arid', 'sere', 'parch', 'dryer', 'wring', 'drear', 'sear', 'pall', 'lack']]

Model response:
The successor function failed to produce all successors because it only tries to place words horizontally from left to right and vertically from top to bottom. It doesn't try to place words in other directions or in other positions.
Here's the corrected successor function:
Python
```
def successor_state(state, horizontal_answers, vertical_answers):
```

````
    def is_valid_word(word, row, col, direction):
        if direction == 'h' and col + len(word) > 5:
            return False
        if direction == 'v' and row + len(word) > 5:
            return False
        for i in range(len(word)):
            if direction == 'h' and (state[row][col + i] is not None
                and state[row][col + i] != word[i]):
                return False
            if direction == 'v' and (state[row + i][col] is not None
                and state[row + i][col] != word[i]):
                return False
        return True

    def place_word(word, row, col, direction):
        new_state = [r[:] for r in state]
        for i in range(len(word)):
            if direction == 'h':
                new_state[row][col + i] = word[i]
            else:
                new_state[row + i][col] = word[i]
        return new_state

    successors = []
    for row in range(5):
        for col in range(5):
            if state[row][col] is None:
                for word in horizontal_answers[row]:
                    if is_valid_word(word, row, col, 'h'):
                        successors.append(place_word(word, row, col, '
                            h'))
                for word in vertical_answers[col]:
                    if is_valid_word(word, row, col, 'v'):
                        successors.append(place_word(word, row, col, '
                            v'))
    return successors
```
In the corrected version, I've added a direction parameter to the
    is_valid_word and place_word functions to handle both horizontal
    and vertical word placement. I've also added a check to ensure
    that the word doesn't exceed the grid boundaries when placing it
    vertically

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 20, failure: 0, total expansions 4514757, total generated
    16830232
````

9.3.5 Llama3.1 405B

Listing 44: Without partial successor soundness test

```
AutoToS prompt:

The task is a 5x5 mini crosswords. A state is a 2D array representing
    the current puzzle state, where the initial grid is all "None".
    Note that some of the possible answers are not exactly 5 character
     long. Given an input of possible answers to horizontal clues and
    vertical clues, how is the successor state function defined?
    Please first think step by step. Then provide the successor state
    function in Python code.
```
```

```
The possible clues for each row and each column are given separately.
    Here is an example of possible horizontal and vertical clues:

horizontal_answers = [
        ["tasks", "goals", "plans", "agend", "chores", "works", "deeds
            ", "items", "lists", "brief"],
        ["motor", "power", "drive", "diesel", "steam", "pumps", "crank
            ", "gears", "turbn", "motor"],
        ["grand", "artsy", "showy", "ornate", "fancy", "vain", "proud
            ", "vogue", "swank", "luxus"],
        ["venue", "salle", "forum", "atria", "lobby", "parls", "court
            ", "malls", "mall", "lobby"],
        ["jeer", "scoff", "sleer", "deris", "sneer", "scorn", "derid",
             "gibes", "gibed", "flout"]
]
vertical_answers = [
        ["amass", "stack", "hoard", "pile", "store", "heaps", "massy",
             "gathe", "lumps", "mound"],
        ["nilga", "goral", "eland", "lepus", "gazal", "kudu", "oryx",
            "gnu", "imps", "carb"],
        ["scheme", "design", "ettle", "nettle", "sting", "wiles", "
            plans", "ideas", "plots", "cocks"],
        ["spout", "nosle", "snout", "mouth", "nostr", "ports", "inlet
            ", "vents", "outlt", "beaks"],
        ["drier", "arid", "sere", "parch", "dryer", "wring", "drear",
            "sear", "pall", "lack"]
]

where horizontal_answers is a list where element i is a list of
    possible answers to clue in row i, and vertical_answers is a list
    where element i is a list of possible answers to clue in column i.

Model response:
Successor State Function for 5x5 Mini Crosswords
```Python
def successor_state(state, horizontal_answers, vertical_answers):
    """
    Generates all possible successor states for the given 5x5 mini
        crossword state.

    Args:
    state (list of lists): 2D array representing the current puzzle
        state.
    horizontal_answers (list of lists): Possible answers to horizontal
         clues.
    vertical_answers (list of lists): Possible answers to vertical
        clues.

    Yields:
    list of lists: A possible successor state.
    """
    def is_valid(state, row, col, answer, direction):
        # Check if the answer can be placed at the given position
        if direction == 'h':
            if col + len(answer) > 5:
                return False
            for i, char in enumerate(answer):
                if state[row][col + i] is not None and state[row][col
                    + i] != char:
                    return False
        else:
            if row + len(answer) > 5:
                return False
            for i, char in enumerate(answer):
```
```

```
                if state[row + i][col] is not None and state[row + i][
                    col] != char:
                    return False
        return True

    def place_answer(state, row, col, answer, direction):
        # Place the answer at the given position
        new_state = [row[:] for row in state]
        if direction == 'h':
            for i, char in enumerate(answer):
                new_state[row][col + i] = char
        else:
            for i, char in enumerate(answer):
                new_state[row + i][col] = char
        return new_state

    # Generate all possible successor states
    for row in range(5):
        for col in range(5):
            if state[row][col] is None:
                # Try to place horizontal answers
                for answer in horizontal_answers[row]:
                    if is_valid(state, row, col, answer, 'h'):
                        yield place_answer(state, row, col, answer, 'h
                            ')
                # Try to place vertical answers
                for answer in vertical_answers[col]:
                    if is_valid(state, row, col, answer, 'v'):
                        yield place_answer(state, row, col, answer, 'v
                            ')
```
This function generates all possible successor states by trying to place each possible answer at each empty position in the grid. It checks if the answer can be placed at the given position without conflicting with existing letters, and if so, yields the resulting state. The is_valid function checks if an answer can be placed at a given position, and the place_answer function creates a new state with the answer placed at the given position.
Generated by Meta Llama 3.1-405B

AutoToS prompt:

Please provide a goal test function in Python ensuring that all cells are not None, all words horizontally are matching at least one of the horizontal answers, and all words vertically match at least one of the vertical answers. The function should receive a state, horizontal_answers, and vertical_answers as before and return True for a goal state and False otherwise.

Model response:
Goal Test Function for 5x5 Mini Crosswords
```Python
def is_goal_state(state, horizontal_answers, vertical_answers):
    """
    Checks if the given state is a goal state.

    Args:
    state (list of lists): 2D array representing the current puzzle
        state.
    horizontal_answers (list of lists): Possible answers to horizontal
        clues.
    vertical_answers (list of lists): Possible answers to vertical
        clues.

    Returns:
```

```
    bool: True if the state is a goal state, False otherwise.
    """
    def is_valid_word(word, answers):
        # Check if the word matches at least one of the answers
        return any(len(word) == len(answer) and all(a == b for a, b in
             zip(word, answer)) for answer in answers)

    # Check if all cells are not None
    if any(cell is None for row in state for cell in row):
        return False

    # Check horizontal words
    for row in range(5):
        word = ''.join(state[row])
        if not is_valid_word(word, horizontal_answers[row]):
            return False

    # Check vertical words
    for col in range(5):
        word = ''.join(state[row][col] for row in range(5))
        if not is_valid_word(word, vertical_answers[col]):
            return False

    return True
```
```
This function first checks if all cells in the state are not None.
    Then, it checks each horizontal word and each vertical word to
    ensure that they match at least one of the corresponding answers.
    If all checks pass, it returns True, indicating that the state is
    a goal state. Otherwise, it returns False. The is_valid_word
    function checks if a word matches at least one of the answers by
    comparing the word with each answer.
Generated by Meta Llama 3.1-405B

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 20, failure: 0, total expansions 4514757, total generated
    16830232
```

Listing 45: With partial successor soundness test

```
AutoToS prompt:

The task is a 5x5 mini crosswords. A state is a 2D array representing
    the current puzzle state, where the initial grid is all "None".
    Note that some of the possible answers are not exactly 5 character
     long. Given an input of possible answers to horizontal clues and
    vertical clues, how is the successor state function defined?
    Please first think step by step. Then provide the successor state
    function in Python code.

The possible clues for each row and each column are given separately.
    Here is an example of possible horizontal and vertical clues:

horizontal_answers = [
        ["tasks", "goals", "plans", "agend", "chores", "works", "deeds
            ", "items", "lists", "brief"],
```

```
        ["motor", "power", "drive", "diesel", "steam", "pumps", "crank
            ", "gears", "turbn", "motor"],
        ["grand", "artsy", "showy", "ornate", "fancy", "vain", "proud
            ", "vogue", "swank", "luxus"],
        ["venue", "salle", "forum", "atria", "lobby", "parls", "court
            ", "malls", "mall", "lobby"],
        ["jeer", "scoff", "sleer", "deris", "sneer", "scorn", "derid",
             "gibes", "gibed", "flout"]
]
vertical_answers = [
        ["amass", "stack", "hoard", "pile", "store", "heaps", "massy",
             "gathe", "lumps", "mound"],
        ["nilga", "goral", "eland", "lepus", "gazal", "kudu", "oryx",
            "gnu", "imps", "carb"],
        ["scheme", "design", "ettle", "nettle", "sting", "wiles", "
            plans", "ideas", "plots", "cocks"],
        ["spout", "nosle", "snout", "mouth", "nostr", "ports", "inlet
            ", "vents", "outlt", "beaks"],
        ["drier", "arid", "sere", "parch", "dryer", "wring", "drear",
            "sear", "pall", "lack"]
]
```

where horizontal_answers is a list where element i is a list of possible answers to clue in row i, and vertical_answers is a list where element i is a list of possible answers to clue in column i.

Model response:
Successor State Function for 5x5 Mini Crosswords
Step-by-Step Thought Process
To define the successor state function, we need to consider the possible next states given a current state and the possible answers to horizontal and vertical clues. Here’s a step-by-step breakdown:
Current State: The current state is a 2D array (5x5 grid) where each cell can be either None (empty) or a character (part of a word).
Possible Next States: For each empty cell in the current state, we need to consider all possible characters that can fill that cell based on the given horizontal and vertical clues.
Horizontal Clues: For each row, we have a list of possible answers. We need to check which of these answers can be placed in the current row, considering the already filled cells.
Vertical Clues: Similarly, for each column, we have a list of possible answers. We need to check which of these answers can be placed in the current column, considering the already filled cells.
Successor State: A successor state is a new 2D array where one empty cell is filled with a character based on the possible answers to horizontal and vertical clues.
Python Code for Successor State Function
Python
```
def successor_state(current_state, horizontal_answers,
    vertical_answers):
    """
    Returns a list of possible successor states given the current
        state and possible answers to horizontal and vertical clues.

    Args:
        current_state (list of lists): 2D array representing the
            current puzzle state.
        horizontal_answers (list of lists): List of possible answers
            to horizontal clues for each row.
        vertical_answers (list of lists): List of possible answers to
            vertical clues for each column.

    Returns:
```

```
        list of lists of lists: List of possible successor states.
    """
    successor_states = []
    for i in range(5):  # Iterate over rows
        for j in range(5):  # Iterate over columns
            if current_state[i][j] is None:  # Check if cell is empty
                # Check horizontal clues
                for answer in horizontal_answers[i]:
                    if len(answer) <= 5 - j:  # Check if answer can
                        fit in the row
                        new_state = [row[:] for row in current_state]
                             # Create a copy of the current state
                        for k, char in enumerate(answer):
                            new_state[i][j + k] = char
                        successor_states.append(new_state)

                # Check vertical clues
                for answer in vertical_answers[j]:
                    if len(answer) <= 5 - i:  # Check if answer can
                        fit in the column
                        new_state = [row[:] for row in current_state]
                             # Create a copy of the current state
                        for k, char in enumerate(answer):
                            new_state[i + k][j] = char
                        successor_states.append(new_state)
    return successor_states
```
This function returns a list of possible successor states, where each successor state is a new 2D array with one empty cell filled with a character based on the possible answers to horizontal and vertical clues. Note that this implementation assumes that the input current state, horizontal answers, and vertical answers are valid and consistent with the problem description.
Generated by Meta Llama 3.1-405

AutoToS prompt:

Please provide a goal test function in Python ensuring that all cells are not None, all words horizontally are matching at least one of the horizontal answers, and all words vertically match at least one of the vertical answers. The function should receive a state, horizontal_answers, and vertical_answers as before and return True for a goal state and False otherwise.

Model response:
Goal Test Function for 5x5 Mini Crosswords
Python
```
def goal_test(state, horizontal_answers, vertical_answers):
    """
    Checks if the given state is a goal state.

    A goal state is a state where:
    1. All cells are not None.
    2. All words horizontally match at least one of the horizontal
        answers.
    3. All words vertically match at least one of the vertical answers
        .

    Args:
        state (list of lists): 2D array representing the current
            puzzle state.
        horizontal_answers (list of lists): List of possible answers
            to horizontal clues for each row.
```

```
        vertical_answers (list of lists): List of possible answers to
            vertical clues for each column.

    Returns:
        bool: True if the state is a goal state, False otherwise.
    """
    # Check if all cells are not None
    if any(cell is None for row in state for cell in row):
        return False

    # Check horizontal answers
    for i, row in enumerate(state):
        row_str = ''.join(row)
        if not any(row_str == answer for answer in horizontal_answers[
            i]):
            return False

    # Check vertical answers
    for j in range(5):
        col_str = ''.join(state[i][j] for i in range(5))
        if not any(col_str == answer for answer in vertical_answers[j
            ]):
            return False

    # If all checks pass, the state is a goal state
    return True
```

This function first checks if all cells in the state are not None. Then, it checks if each row matches at least one of the horizontal answers. Finally, it checks if each column matches at least one of the vertical answers. If all checks pass, the function returns True, indicating that the state is a goal state. Otherwise, it returns False.

You can use this function to test if a given state is a goal state, like this:

Python

```
state = [
    ['t', 'a', 's', 'k', 's'],
    ['m', 'o', 't', 'o', 'r'],
    ['g', 'r', 'a', 'n', 'd'],
    ['v', 'e', 'n', 'u', 'e'],
    ['j', 'e', 'e', 'r', 's']
]

horizontal_answers = [
    ["tasks", "goals", "plans", "agend", "chores", "works", "deeds", "
        items", "lists", "brief"],
    ["motor", "power", "drive", "diesel", "steam", "pumps", "crank", "
        gears", "turbn", "motor"],
    ["grand", "artsy", "showy", "ornate", "fancy", "vain", "proud", "
        vogue", "swank", "luxus"],
    ["venue", "salle", "forum", "atria", "lobby", "parls", "court", "
        malls", "mall", "lobby"],
    ["jeer", "scoff", "sleer", "deris", "sneer", "scorn", "derid", "
        gibes", "gibed", "flout"]
]

vertical_answers = [
    ["amass", "stack", "hoard", "pile", "store", "heaps", "massy", "
        gathe", "lumps", "mound"],
    ["nilga", "goral", "eland", "lepus", "gazal", "kudu", "oryx", "gnu
        ", "imps", "carb"],
    ["scheme", "design", "ettle", "nettle", "sting", "wiles", "plans",
         "ideas", "plots", "cocks"],
```

```
    ["spout", "nosle", "snout", "mouth", "nostr", "ports", "inlet", "
        vents", "outlt", "beaks"],
    ["drier", "arid", "sere", "parch", "dryer", "wring", "drear", "
        sear", "pall", "lack"]
]

print(goal_test(state, horizontal_answers, vertical_answers))  #
    Output: True
Generated by Meta Llama 3.1-405

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Evaluating the final outcome
Success: 20, failure: 0, total expansions 2584134, total generated
    6255012
```

9.4 ProntoQA

9.4.1 Deepseek-CoderV2

Listing 46: Without partial successor soundness test

```
AutoToS prompt:

Logical deduction is performed by applying rules of the form if X then
     Y to our current knowledge about a world, extending our knowledge
     with each single rule application.
A rule if X then Y is applicable if we know X. If a state is defined
    as our current knowledge of the world, how are successor states
    defined, given a collection of rules? Please think step by step.
    Then, provide only the Python implementation for the successor
    function that receives a state and rules and returns a list of
    successor states.

Example initial state: {"integer"}
Example rules: [("integer", "real number"), ("integer", "float")]

Model response:
To define successor states given a collection of rules, we need to
    follow these steps:

1. **Identify Applicable Rules**: For each rule `if X then Y`, check
    if `X` is present in the current state.
2. **Generate Successor States**: For each applicable rule, create a
    new state by adding `Y` to the current state.
3. **Ensure Uniqueness**: Each successor state should be unique to
    avoid redundant states.

Let's break down the process step-by-step:

1. **Identify Applicable Rules**:
   - Iterate through the list of rules.
   - For each rule `(X, Y)`, check if `X` is in the current state.

2. **Generate Successor States**:
   - If `X` is in the current state, create a new state by copying the
       current state and adding `Y` to it.

3. **Ensure Uniqueness**:
```

~~~~
   - Use a set or a similar data structure to ensure that each
       successor state is unique.

Now, let’s implement this in Python:

python
```
def successor(state, rules):
    successor_states = []
    for rule in rules:
        X, Y = rule
        if X in state:
            new_state = state.copy()
            new_state.add(Y)
            if new_state not in successor_states:
                successor_states.append(new_state)
    return successor_states

# Example usage:
initial_state = {"integer"}
rules = [("integer", "real number"), ("integer", "float")]
print(successor(initial_state, rules))
```

### Explanation:
- **state**: A set representing the current knowledge of the world.
- **rules**: A list of tuples where each tuple `(X, Y)` represents a
    rule `if X then Y`.
- **successor_states**: A list to store unique successor states.
- **for rule in rules**: Iterate through each rule.
- **if X in state**: Check if `X` is in the current state.
- **new_state = state.copy()**: Create a copy of the current state.
- **new_state.add(Y)**: Add `Y` to the new state.
- **if new_state not in successor_states**: Ensure the new state is
    unique before adding it to the list of successor states.

This function will return a list of unique successor states based on
    the given rules and the current state

AutoToS prompt:

Provide only the Python function for the goal test function that tests
    , given a target fact, whether a state is a goal state.

Model response:
python
```
def goal_test(state, target_fact):
    return target_fact in state
``

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 4000, failure: 0, total expansions 85523, total evaluations
    244776
~~~~

Listing 47: With partial successor soundness test

~~~~
AutoToS prompt:
~~~~

Logical deduction is performed by applying rules of the form if X then Y to our current knowledge about a world, extending our knowledge with each single rule application.
A rule if X then Y is applicable if we know X. If a state is defined as our current knowledge of the world, how are successor states defined, given a collection of rules? Please think step by step. Then, provide only the Python implementation for the successor function that receives a state and rules and returns a list of successor states.

Example initial state: {"integer"}
Example rules: [("integer", "real number"), ("integer", "float")]

Model response:
To define the successor states given a collection of rules, we need to follow these steps:

Identify Applicable Rules: For each rule if X then Y, check if X is present in the current state.

Generate Successor States: For each applicable rule, create a new state by adding Y to the current state.

Ensure Uniqueness: Each successor state should be unique to avoid redundant states.

Let's break down the process step-by-step:

Identify Applicable Rules:

Iterate through the list of rules.

For each rule (X, Y), check if X is in the current state.

Generate Successor States:

If X is in the current state, create a new state by copying the current state and adding Y to it.

Ensure Uniqueness:

Use a set or a similar data structure to ensure that each successor state is unique.

Now, let's implement the successor function in Python:

python
Copy code
```
def successor(state, rules):
    successor_states = []
    for rule in rules:
        X, Y = rule
        if X in state:
            new_state = state.copy()
            new_state.add(Y)
            if new_state not in successor_states:
                successor_states.append(new_state)
    return successor_states

# Example usage:
initial_state = {"integer"}
rules = [("integer", "real number"), ("integer", "float")]
print(successor(initial_state, rules))
```

```
```
Explanation:
state: A set representing the current knowledge of the world.

rules: A list of tuples where each tuple (X, Y) represents a rule if X
     then Y.

successor_states: A list to store the unique successor states.

for rule in rules: Iterate through each rule.

if X in state: Check if X is in the current state.

new_state = state.copy(): Create a copy of the current state.

new_state.add(Y): Add Y to the new state.

if new_state not in successor_states: Ensure the new state is unique
    before adding it to the list of successor states.

This function will return a list of unique successor states based on
    the given rules and the current state

AutoToS prompt:

Provide only the Python function for the goal test function that tests
    , given a target fact, whether a state is a goal state.

Model response:
Here is the Python function for the goal test function:

python
Copy code
```
def is_goal_state(state, target_fact):
    return target_fact in state

# Example usage:
# state = {"integer", "real number"}
# target_fact = "real number"
# print(is_goal_state(state, target_fact))  # Output: True
```
This function checks if the target_fact is present in the state and
    returns True if it is, indicating that the state is a goal state.
    Otherwise, it returns False

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 4000, failure: 0, total expansions 86963, total evaluations
    248574
```

#### 9.4.2 GPT4-Mini

Listing 48: Without partial successor soundness test

```
AutoToS prompt:
```
```

```
Logical deduction is performed by applying rules of the form if X then
     Y to our current knowledge about a world, extending our knowledge
     with each single rule application.
A rule if X then Y is applicable if we know X. If a state is defined
    as our current knowledge of the world, how are successor states
    defined, given a collection of rules? Please think step by step.
    Then, provide only the Python implementation for the successor
    function that receives a state and rules and returns a list of
    successor states.

Example initial state: {"integer"}
Example rules: [("integer", "real number"), ("integer", "float")]

Model response:
```python
def successor(state, rules):
    def apply_rules(state, rules):
        successors = set()
        for x, y in rules:
            if x in state:
                successors.add(y)
        return successors

    return list(apply_rules(state, rules))
```

AutoToS prompt:

Provide only the Python function for the goal test function that tests
    , given a target fact, whether a state is a goal state.

Model response:
```python
def is_goal_state(state, target_fact):
    return target_fact in state
```

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 1

AutoToS prompt:
Successor function when run on the state {'painted lady'} failed to
    produce all successors.
Missing successors are: [{'painted lady', 'butterfly'}]
First think step by step why the successor function failed to produce
    all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.
Rules: [('arthropod', 'protostome'), ('lepidopteran', 'insect'), ('
    painted lady', 'butterfly'), ('insect', 'arthropod'), ('
    invertebrate', 'animal'), ('arthropod', 'not-bony'), ('protostome
    ', 'invertebrate'), ('whale', 'bony'), ('butterfly', 'lepidopteran
    '), ('animal', 'multicellular'), ('insect', 'six-legged')]

Model response:
```python
def successor(state, rules):
    def apply_rules(state, rules):
```
```

```
        successors = set(state)  # Start with current state in
            successors
        changes = True

        while changes:  # Continue until no new successors are found
            changes = False
            new_successors = set()
            for x, y in rules:
                if x in successors and y not in successors:
                    new_successors.add(y)
                    changes = True
            successors.update(new_successors)

        return successors

    return list(apply_rules(state, rules))
```

```
System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 2

AutoToS prompt:
Successor function when run on the state {'painted lady'} failed to
    produce all successors.
Missing successors are: [{'painted lady', 'butterfly'}]
First think step by step why the successor function failed to produce
    all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.
Rules: [('arthropod', 'protostome'), ('lepidopteran', 'insect'), ('
    painted lady', 'butterfly'), ('insect', 'arthropod'), ('
    invertebrate', 'animal'), ('arthropod', 'not-bony'), ('protostome
    ', 'invertebrate'), ('whale', 'bony'), ('butterfly', 'lepidopteran
    '), ('animal', 'multicellular'), ('insect', 'six-legged')]

Model response:
```

```python
def successor(state, rules):
    def apply_rules(state, rules):
        successors = set(state)  # Start with the current state in
            successors
        changes = True

        while changes:  # Continue until no new successors are found
            changes = False
            new_successors = set()
            for x, y in rules:
                if x in successors and y not in successors:
                    new_successors.add(y)
                    changes = True
            successors.update(new_successors)

        return successors

    return [tuple(successor) for successor in apply_rules(state, rules
        )]
```

```
System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 3
```

```
AutoToS prompt:
Successor function when run on the state {'painted lady'} failed to
    produce all successors.
Missing successors are: [{'painted lady', 'butterfly'}]
First think step by step why the successor function failed to produce
    all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.
Rules: [('arthropod', 'protostome'), ('lepidopteran', 'insect'), ('
    painted lady', 'butterfly'), ('insect', 'arthropod'), ('
    invertebrate', 'animal'), ('arthropod', 'not-bony'), ('protostome
    ', 'invertebrate'), ('whale', 'bony'), ('butterfly', 'lepidopteran
    '), ('animal', 'multicellular'), ('insect', 'six-legged')]

Model response:
```python
def successor(state, rules):
    def apply_rules(state, rules):
        successors = set(state)  # Start with the current state in
            successors
        changes = True

        while changes:  # Continue until no new successors are found
            changes = False
            new_successors = set()
            for x, y in rules:
                if x in successors and y not in successors:
                    new_successors.add(y)
                    changes = True
            successors.update(new_successors)

        return successors

    initial_successors = apply_rules(state, rules)
    all_successors = [{*state, s} for s in initial_successors]  #
        Include all initial states with each successor
    return list(all_successors)
```

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 4000, failure: 0, total expansions 897073, total evaluations
    9156989
```

Listing 49: With partial successor soundness test

```
AutoToS prompt:

Logical deduction is performed by applying rules of the form if X then
     Y to our current knowledge about a world, extending our knowledge
     with each single rule application.
A rule if X then Y is applicable if we know X. If a state is defined
    as our current knowledge of the world, how are successor states
    defined, given a collection of rules? Please think step by step.
    Then, provide only the Python implementation for the successor
    function that receives a state and rules and returns a list of
    successor states.

Example initial state: {"integer"}
Example rules: [("integer", "real number"), ("integer", "float")]
```

Model response:
```python
def successor(state, rules):
    def apply_rules(state, rules):
        successors = set()
        for rule in rules:
            if rule[0] in state:
                successors.add(rule[1])
        return successors

    current_successors = apply_rules(state, rules)
    return list(current_successors)
```

AutoToS prompt:

Provide only the Python function for the goal test function that tests, given a target fact, whether a state is a goal state.

Model response:
```python
def is_goal_state(state, target_fact):
    return target_fact in state
```

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
Invalid transition: length mismatch - the length of a successor must be one more than the parent.
Let's think step by step. First think through in words why the successor function produced a successor that had a length that was not exactly one more than the parent. Then provide the complete Python code for the revised successor function that ensures the length of a successor is exactly one more than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: {'painted lady'}
Example wrong successor state: butterfly
Rules: {('butterfly', 'lepidopteran'), ('lepidopteran', 'insect'), ('arthropod', 'not-bony'), ('animal', 'multicellular'), ('whale', 'bony'), ('insect', 'arthropod'), ('invertebrate', 'animal'), ('insect', 'six-legged'), ('painted lady', 'butterfly'), ('protostome', 'invertebrate'), ('arthropod', 'protostome')}

Model response:
```python
def successor(state, rules):
    def apply_rules(state, rules):
        successors = set()
        for rule in rules:
            if rule[0] in state:
                successors.add(rule[1])
        return successors

    current_successors = apply_rules(state, rules)
    valid_successors = [s for s in current_successors if len(state) +
        1 == len({*state, s})]

    return valid_successors
```

Successor Iteration 2

AutoToS prompt:
Invalid transition: length mismatch – the length of a successor must be one more than the parent.
Let’s think step by step. First think through in words why the successor function produced a successor that had a length that was not exactly one more than the parent. Then provide the complete Python code for the revised successor function that ensures the length of a successor is exactly one more than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: {’painted lady’}
Example wrong successor state: butterfly
Rules: {(’butterfly’, ’lepidopteran’), (’lepidopteran’, ’insect’), (’arthropod’, ’not-bony’), (’animal’, ’multicellular’), (’whale’, ’bony’), (’insect’, ’arthropod’), (’invertebrate’, ’animal’), (’insect’, ’six-legged’), (’painted lady’, ’butterfly’), (’protostome’, ’invertebrate’), (’arthropod’, ’protostome’)}

Model response:
```python
def successor(state, rules):
    def apply_rules(state, rules):
        new_successors = set()
        for rule in rules:
            if rule[0] in state:
                new_successors.add(rule[1])
        return new_successors

    current_successors = apply_rules(state, rules)

    valid_successors = []
    for successor in current_successors:
        if len(state) + 1 == len({*state, successor}):
            valid_successors.append(successor)

    return valid_successors
```
Successor Iteration 3

AutoToS prompt:
Invalid transition: length mismatch – the length of a successor must be one more than the parent.
Let’s think step by step. First think through in words why the successor function produced a successor that had a length that was not exactly one more than the parent. Then provide the complete Python code for the revised successor function that ensures the length of a successor is exactly one more than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: {’painted lady’}
Example wrong successor state: butterfly
Rules: {(’butterfly’, ’lepidopteran’), (’lepidopteran’, ’insect’), (’arthropod’, ’not-bony’), (’animal’, ’multicellular’), (’whale’, ’bony’), (’insect’, ’arthropod’), (’invertebrate’, ’animal’), (’insect’, ’six-legged’), (’painted lady’, ’butterfly’), (’protostome’, ’invertebrate’), (’arthropod’, ’protostome’)}

Model response:
```python
def successor(state, rules):
    def apply_rules(state, rules):
```

```
        new_successors = set()
        for rule in rules:
            if rule[0] in state:
                new_successors.add(rule[1])
        return new_successors

    current_successors = apply_rules(state, rules)

    valid_successors = []
    for successor in current_successors:
        # Check that the successor is a new fact and the length
            condition is satisfied
        if successor not in state and len(state) + 1 == len(state | {
            successor}):
            valid_successors.append(successor)

    return valid_successors
```

Successor Iteration 4

AutoToS prompt:
Invalid transition: length mismatch - the length of a successor must be one more than the parent.
Let's think step by step. First think through in words why the successor function produced a successor that had a length that was not exactly one more than the parent. Then provide the complete Python code for the revised successor function that ensures the length of a successor is exactly one more than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: {'painted lady'}
Example wrong successor state: butterfly
Rules: {('butterfly', 'lepidopteran'), ('lepidopteran', 'insect'), ('arthropod', 'not-bony'), ('animal', 'multicellular'), ('whale', 'bony'), ('insect', 'arthropod'), ('invertebrate', 'animal'), ('insect', 'six-legged'), ('painted lady', 'butterfly'), ('protostome', 'invertebrate'), ('arthropod', 'protostome')}

Model response:
```python
def successor(state, rules):
    def apply_rules(state, rules):
        new_successors = set()
        for rule in rules:
            if rule[0] in state:
                new_successors.add(rule[1])
        return new_successors

    current_successors = apply_rules(state, rules)

    valid_successors = []
    current_state_length = len(state)

    for successor in current_successors:
        # Check that the successor is a new fact and that the
            resulting state has the correct length
        if successor not in state and len(state | {successor}) ==
            current_state_length + 1:
            valid_successors.append(successor)

    return valid_successors
```

Successor Iteration 5

AutoToS prompt:
Invalid transition: length mismatch - the length of a successor must
    be one more than the parent.
Let's think step by step. First think through in words why the
    successor function produced a successor that had a length that was
     not exactly one more than the parent. Then provide the complete
    Python code for the revised successor function that ensures the
    length of a successor is exactly one more than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.

Input state: {'painted lady'}
Example wrong successor state: butterfly
Rules: {('butterfly', 'lepidopteran'), ('lepidopteran', 'insect'), ('
    arthropod', 'not-bony'), ('animal', 'multicellular'), ('whale', '
    bony'), ('insect', 'arthropod'), ('invertebrate', 'animal'), ('
    insect', 'six-legged'), ('painted lady', 'butterfly'), ('
    protostome', 'invertebrate'), ('arthropod', 'protostome')}

Model response:
```python
def successor(state, rules):
    def apply_rules(state, rules):
        new_successors = set()
        for rule in rules:
            if rule[0] in state:
                new_successors.add(rule[1])
        return new_successors

    current_successors = apply_rules(state, rules)

    valid_successors = []
    current_state_length = len(state)

    for successor in current_successors:
        # Check that the successor is not already in the state
        if successor not in state:
            # Create a potential new state by adding the successor
            new_state = state | {successor}
            # Validate that the new state's length is exactly one
                greater than the current state's length
            if len(new_state) == current_state_length + 1:
                valid_successors.append(successor)

    return valid_successors
```
Successor Iteration 6

AutoToS prompt:
Invalid transition: length mismatch - the length of a successor must
    be one more than the parent.
Let's think step by step. First think through in words why the
    successor function produced a successor that had a length that was
     not exactly one more than the parent. Then provide the complete
    Python code for the revised successor function that ensures the
    length of a successor is exactly one more than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.

Input state: {'painted lady'}
Example wrong successor state: butterfly
Rules: {('butterfly', 'lepidopteran'), ('lepidopteran', 'insect'), ('
    arthropod', 'not-bony'), ('animal', 'multicellular'), ('whale', '
    bony'), ('insect', 'arthropod'), ('invertebrate', 'animal'), ('

insect', 'six-legged'), ('painted lady', 'butterfly'), ('protostome', 'invertebrate'), ('arthropod', 'protostome')}

Model response:

```python
def successor(state, rules):
    def apply_rules(state, rules):
        new_successors = set()
        for rule in rules:
            if rule[0] in state:
                new_successors.add(rule[1])
        return new_successors

    current_successors = apply_rules(state, rules)

    valid_successors = []
    current_state_length = len(state)

    for successor in current_successors:
        # Check that the successor is not already in the state
        if successor not in state:
            # Create a potential new state by adding the successor
            new_state = state | {successor}
            # Validate that the new state's length is exactly one
                greater than the current state's length
            if len(new_state) == current_state_length + 1:
                valid_successors.append(successor)

    return valid_successors
```

Successor Iteration 7

AutoToS prompt:
Invalid transition: length mismatch - the length of a successor must be one more than the parent.
Let's think step by step. First think through in words why the successor function produced a successor that had a length that was not exactly one more than the parent. Then provide the complete Python code for the revised successor function that ensures the length of a successor is exactly one more than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: {'painted lady'}
Example wrong successor state: butterfly
Rules: {('butterfly', 'lepidopteran'), ('lepidopteran', 'insect'), ('arthropod', 'not-bony'), ('animal', 'multicellular'), ('whale', 'bony'), ('insect', 'arthropod'), ('invertebrate', 'animal'), ('insect', 'six-legged'), ('painted lady', 'butterfly'), ('protostome', 'invertebrate'), ('arthropod', 'protostome')}

Model response:
The reason the successor function may produce a successor that does not maintain the required length (one more than the parent state) is because it might allow for adding a state that is already present or inadvertently include rules that do not extend the state by exactly one new fact.

To ensure that the successor state length is exactly one more than the parent state, we need to implement a stricter check when applying the rules. Specifically, we will generate a new set that includes the current state and one additional fact resulting from the rules.

Here is the corrected code for the successor function:

```python
def successor(state, rules):
    def apply_rules(state, rules):
        new_successors = set()
        for rule in rules:
            if rule[0] in state:
                new_successors.add(rule[1])
        return new_successors

    current_successors = apply_rules(state, rules)

    valid_successors = []

    for successor in current_successors:
        # Ensure that the successor is not already in the state
        if successor not in state:
            # Create a potential new state by adding the successor to
                the current state
            new_state = state | {successor}
            # Check that the new state's length is exactly one greater
                 than the current state's length
            if len(new_state) == len(state) + 1:
                valid_successors.append(successor)

    return valid_successors
```

This revision ensures that any valid successor produced results in a set whose length is one greater than the original state, thus fulfilling the required condition.
Successor Iteration 8

AutoToS prompt:
Invalid transition: length mismatch - the length of a successor must be one more than the parent.
Let's think step by step. First think through in words why the successor function produced a successor that had a length that was not exactly one more than the parent. Then provide the complete Python code for the revised successor function that ensures the length of a successor is exactly one more than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: {'painted lady'}
Example wrong successor state: butterfly
Rules: {('butterfly', 'lepidopteran'), ('lepidopteran', 'insect'), ('arthropod', 'not-bony'), ('animal', 'multicellular'), ('whale', 'bony'), ('insect', 'arthropod'), ('invertebrate', 'animal'), ('insect', 'six-legged'), ('painted lady', 'butterfly'), ('protostome', 'invertebrate'), ('arthropod', 'protostome')}

Model response:
The core issue causing the length mismatch arises when the successor can potentially introduce facts that are either already contained within the state or do not create a distinct new state. Our goal is to ensure that each successor state maintains exactly one more fact compared to its parent state, hence requiring precise validation.

Here is the complete Python code for the revised successor function:

```python
def successor(state, rules):
    def apply_rules(state, rules):
```

```
        new_successors = set()
        for rule in rules:
            if rule[0] in state:
                new_successors.add(rule[1])
        return new_successors

    current_successors = apply_rules(state, rules)

    valid_successors = []

    for successor in current_successors:
        # We must check that the successor is a new fact
        if successor not in state:
            # Create a new set that combines the current state with
                the successor
            new_state = state | {successor}
            # Validate that the new state's length is exactly one more
                 than the original
            if len(new_state) == len(state) + 1:
                valid_successors.append(successor)

    return valid_successors
```

In this function, we apply the rules to find potential successors, and then we check whether including each successor in the state results in a new state that is precisely one fact longer than the original. This ensures compliance with the requirement of the successor function.
Successor Iteration 9

AutoToS prompt:
Invalid transition: length mismatch - the length of a successor must be one more than the parent.
Let's think step by step. First think through in words why the successor function produced a successor that had a length that was not exactly one more than the parent. Then provide the complete Python code for the revised successor function that ensures the length of a successor is exactly one more than the parent.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: {'painted lady'}
Example wrong successor state: butterfly
Rules: {('butterfly', 'lepidopteran'), ('lepidopteran', 'insect'), ('arthropod', 'not-bony'), ('animal', 'multicellular'), ('whale', 'bony'), ('insect', 'arthropod'), ('invertebrate', 'animal'), ('insect', 'six-legged'), ('painted lady', 'butterfly'), ('protostome', 'invertebrate'), ('arthropod', 'protostome')}

Model response:
The requirement for the successor function is that a valid successor state must have a length exactly one greater than the parent state. The previous implementations may have produced successors that were not unique to the state or that did not result in a distinct increase in the fact count.

To ensure each successor adds exactly one new fact, we should clarify our filtering by confirming that the successor is contributed by the rules and that it was not already included in the current state.

Here's the revised implementation:

```python
```

```
def successor(state, rules):
    def apply_rules(state, rules):
        new_successors = set()
        for rule in rules:
            if rule[0] in state:
                new_successors.add(rule[1])
        return new_successors

    current_successors = apply_rules(state, rules)

    valid_successors = []

    for successor in current_successors:
        # Ensure that the successor is not already in the state
        if successor not in state:
            # Create a potential new state by adding the successor
            new_state = state | {successor}
            # Check that the new state's length is exactly one greater
                 than the original state's length
            if len(new_state) == len(state) + 1:
                valid_successors.append(successor)

    return valid_successors
```
```

This version checks that:
1. Each successor generated from the rules is indeed new (not already
    part of the original state).
2. The combination of the original state with this new successor
    strictly results in a state that is one fact longer.

By enforcing these conditions, we ensure we comply with the length
    requirement for the successor states.

System message:
Maximum iterations reached for completeness test. Process stopped.
```

9.4.3 GPT4

Listing 50: Without partial successor soundness test

```
AutoToS prompt:

Logical deduction is performed by applying rules of the form if X then
     Y to our current knowledge about a world, extending our knowledge
     with each single rule application.
A rule if X then Y is applicable if we know X. If a state is defined
    as our current knowledge of the world, how are successor states
    defined, given a collection of rules? Please think step by step.
    Then, provide only the Python implementation for the successor
    function that receives a state and rules and returns a list of
    successor states.

Example initial state: {"integer"}
Example rules: [("integer", "real number"), ("integer", "float")]

Model response:
Certainly! The successor function needs to apply each applicable rule
    to the state to generate new successor states. Below is the
    implementation that performs this task.

```python
def successor(state, rules):
    """
```
```

```
    Generate successor states by applying given rules to the current
       state.

    :param state: A set representing the current knowledge about the
       world.
    :param rules: A list of tuples representing rules in the form (X,
       Y) where if X is known, then Y can be deduced.
    :return: A list of sets representing all possible successor states
       .
    """
    successors = []

    for X, Y in rules:
        if X in state:
            new_state = state.copy()
            new_state.add(Y)
            if new_state != state:  # Make sure the new state is
               different from the current state
                successors.append(new_state)

    return successors
```

```
You can use this function by passing your current state and the list
   of rules to it. The function will then return all possible
   successor states that can be derived by applying the rules to the
   current state.

AutoToS prompt:

    Provide only the Python function for the goal test function that
       tests, given a target fact, whether a state is a goal state.

Model response:
Certainly! Below is the function for testing if a state is a goal
   state given a target fact.
```

```python
def is_goal_state(state, target_fact):
    """
    Determine if a given state is a goal state based on the target
       fact.

    :param state: A set representing the current knowledge about the
       world.
    :param target_fact: A fact to check for in the state.
    :return: True if the target fact is in the state, False otherwise.
    """
    return target_fact in state
```

```
System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 4000, failure: 0, total expansions 88141, total evaluations
   176892
```

Listing 51: With partial successor soundness test

```
AutoToS prompt:
```

```
Logical deduction is performed by applying rules of the form if X then
     Y to our current knowledge about a world, extending our knowledge
     with each single rule application.
A rule if X then Y is applicable if we know X. If a state is defined
    as our current knowledge of the world, how are successor states
    defined, given a collection of rules? Please think step by step.
    Then, provide only the Python implementation for the successor
    function that receives a state and rules and returns a list of
    successor states.

Example initial state: {"integer"}
Example rules: [("integer", "real number"), ("integer", "float")]

Model response:
```python
def successor_states(state, rules):
    """
    Generate successor states based on the current state and the
        provided rules.

    :param state: Set of strings representing the current state.
    :param rules: List of tuples where each tuple represents a rule (X
        , Y).
    :return: List of sets representing all successor states.
    """
    successors = []

    for X, Y in rules:
        if X in state:
            new_state = set(state)
            new_state.add(Y)
            if new_state != state:
                successors.append(new_state)

    return successors
```

AutoToS prompt:

Provide only the Python function for the goal test function that tests
    , given a target fact, whether a state is a goal state.

Model response:
```python
def is_goal_state(state, target_fact):
    """
    Test whether the given state is a goal state containing the target
         fact.

    :param state: Set of strings representing the current state.
    :param target_fact: String representing the target fact.
    :return: Boolean indicating whether the state is a goal state.
    """
    return target_fact in state
```

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
```

```
Success: 4000, failure: 0, total expansions 85837, total evaluations
    169779
```

9.4.4 Llama3.1 70B

Listing 52: Without partial successor soundness test

```
AutoToS prompt:

Logical deduction is performed by applying rules of the form if X then
     Y to our current knowledge about a world, extending our knowledge
     with each single rule application.
A rule if X then Y is applicable if we know X. If a state is defined
    as our current knowledge of the world, how are successor states
    defined, given a collection of rules? Please think step by step.
    Then, provide only the Python implementation for the successor
    function that receives a state and rules and returns a list of
    successor states.

Example initial state: {"integer"}
Example rules: [("integer", "real number"), ("integer", "float")]

Model response:
Let's break it down step by step:
Current State: Our current knowledge about the world, represented as a
     set of facts. (e.g., {"integer"})
Rules: A collection of if-then rules, where each rule is a pair of
    facts (e.g., [("integer", "real number"), ("integer", "float")])
Applicable Rules: A rule is applicable if its "if" part (antecedent)
    is present in the current state.
Successor States: For each applicable rule, we create a new state by
    adding the "then" part (consequent) of the rule to the current
    state.
Here's the Python implementation for the successor function:
Python
```
def get_successor_states(state, rules):
    successor_states = []
    for rule in rules:
        if rule[0] in state:
            new_state = state.copy()
            new_state.add(rule[1])
            successor_states.append(new_state)
    return successor_states

# Example usage:
initial_state = {"integer"}
rules = [("integer", "real number"), ("integer", "float")]
successor_states = get_successor_states(initial_state, rules)
print(successor_states) # Output: [{'integer', 'real number'}, {'
    integer', 'float'}]
```
This function iterates over each rule, checks if the antecedent is in
    the current state, and if so, creates a new state by adding the
    consequent to the current state. The resulting list of successor
    states is then returned

AutoToS prompt:

Provide only the Python function for the goal test function that tests
    , given a target fact, whether a state is a goal state.

Model response:
Here is the Python function for the goal test:
Python
```

```
def is_goal_state(state, target_fact):
    return target_fact in state
```
This function checks if the target fact is present in the given state,
     returning True if it is, and False otherwise

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 4000, failure: 0, total expansions 87058, total evaluations
    520385

Listing 53: With partial successor soundness test

AutoToS prompt:

Logical deduction is performed by applying rules of the form if X then
     Y to our current knowledge about a world, extending our knowledge
     with each single rule application.
A rule if X then Y is applicable if we know X. If a state is defined
    as our current knowledge of the world, how are successor states
    defined, given a collection of rules? Please think step by step.
    Then, provide only the Python implementation for the successor
    function that receives a state and rules and returns a list of
    successor states.

Example initial state: {"integer"}
Example rules: [("integer", "real number"), ("integer", "float")]

Model response:
A successor state is a new state that can be reached by applying a
    single rule to the current state. Given a collection of rules, the
     successor states are defined as follows:
Initialize an empty list to store the successor states.
For each rule in the collection of rules:
a. Check if the condition (X) of the rule is present in the current
    state.
b. If the condition is present, create a new state by adding the
    conclusion (Y) of the rule to the current state.
c. Add the new state to the list of successor states.
Return the list of successor states.
Here is the Python implementation for the successor function:
Python
```
def successor_states(state, rules):
    successor_states = []
    for rule in rules:
        if rule[0] in state:
            new_state = state.copy()
            new_state.add(rule[1])
            successor_states.append(new_state)
    return successor_states
```
You can use this function like this:
Python
initial_state = {"integer"}
rules = [("integer", "real number"), ("integer", "float")]
print(successor_states(initial_state, rules))
This will output:

```
Python
[{'integer', 'real number'}, {'integer', 'float'}

AutoToS prompt:

Provide only the Python function for the goal test function that tests
    , given a target fact, whether a state is a goal state.

Model response:
Here is the Python function for the goal test:
Python
```
def goal_test(state, target_fact):
    return target_fact in state
```
This function takes in a state and a target fact, and returns True if
    the target fact is in the state, indicating that the state is a
    goal state. Otherwise, it returns False

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 4000, failure: 0, total expansions 87490, total evaluations
    523778
```

9.4.5 Llama3.1 405B

Listing 54: Without partial successor soundness test

```
AutoToS prompt:

Logical deduction is performed by applying rules of the form if X then
     Y to our current knowledge about a world, extending our knowledge
     with each single rule application.
A rule if X then Y is applicable if we know X. If a state is defined
    as our current knowledge of the world, how are successor states
    defined, given a collection of rules? Please think step by step.
    Then, provide only the Python implementation for the successor
    function that receives a state and rules and returns a list of
    successor states.

Example initial state: {"integer"}
Example rules: [("integer", "real number"), ("integer", "float")]

Model response:
Successor States Definition
Given a state and a collection of rules, we define successor states as
     follows:
Initialize an empty list to store the successor states.
For each rule of the form "if X then Y":
Check if X is present in the current state.
If X is present, create a new state by adding Y to the current state.
Add the new state to the list of successor states.
Return the list of successor states.
Python Implementation
Python
```
def successor_states(state, rules):
    """
    Generate successor states by applying rules to the current state.
```
```

```
    Args:
        state (set): Current knowledge of the world.
        rules (list): List of rules of the form (X, Y), meaning "if X
            then Y".

    Returns:
        list: List of successor states.
    """
    successor_states = []
    for rule in rules:
        X, Y = rule
        if X in state:
            new_state = state.copy()
            new_state.add(Y)
            successor_states.append(new_state)
    return successor_states

# Example usage
initial_state = {"integer"}
rules = [("integer", "real number"), ("integer", "float")]
successors = successor_states(initial_state, rules)
print(successors)
```

This implementation assumes that the state is represented as a set of strings, and the rules are represented as a list of tuples, where each tuple contains two strings representing the condition and the conclusion of the rule. The successor_states function returns a list of sets, where each set represents a successor state.
Generated by Meta Llama 3.1-405

AutoToS prompt:

Provide only the Python function for the goal test function that tests , given a target fact, whether a state is a goal state.

Model response:
Goal Test Function
Python

```
def is_goal_state(state, target_fact):
    """
    Check if a state is a goal state by verifying if it contains the
        target fact.

    Args:
        state (set): Current knowledge of the world.
        target_fact: The fact that must be present in the state to
            consider it a goal state.

    Returns:
        bool: True if the state is a goal state, False otherwise.
    """
    return target_fact in state
```

Generated by Meta Llama 3.1-405

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed

```
Success: 4000, failure: 0, total expansions 87519, total evaluations
    523818
```

Listing 55: With partial successor soundness test

```
AutoToS prompt:

Logical deduction is performed by applying rules of the form if X then
     Y to our current knowledge about a world, extending our knowledge
     with each single rule application.
A rule if X then Y is applicable if we know X. If a state is defined
    as our current knowledge of the world, how are successor states
    defined, given a collection of rules? Please think step by step.
    Then, provide only the Python implementation for the successor
    function that receives a state and rules and returns a list of
    successor states.

Example initial state: {"integer"}
Example rules: [("integer", "real number"), ("integer", "float")]

Model response:
Successor States Definition
Given a state and a collection of rules, a successor state is defined
    as the state extended with the application of one rule. In other
    words, if we have a state S and a rule if X then Y, a successor
    state S' is created by adding Y to S if X is already in S.
Formally, given a state S and a rule if X then Y, a successor state S'
     is defined as:
$S' = S \cup {Y} if X \in S$
In simpler terms, we create a new state by adding the conclusion of
    the rule (Y) to the current state (S) if the premise of the rule (
    X) is already known in the current state.
Python Implementation
Here is the Python implementation of the successor function:
Python
```
```
def successor_states(state, rules):
    """
    Returns a list of successor states given a state and a collection
        of rules.

    Args:
        state (set): The current state of knowledge.
        rules (list of tuples): A list of rules, where each rule is a
            tuple (X, Y) meaning "if X then Y".

    Returns:
        list of sets: A list of successor states.
    """
    successor_states = []
    for rule in rules:
        premise, conclusion = rule
        if premise in state:
            new_state = state.copy()
            new_state.add(conclusion)
            successor_states.append(new_state)
    return successor_states
```
```
You can use this function like this:
Python
initial_state = {"integer"}
rules = [("integer", "real number"), ("integer", "float")]
successors = successor_states(initial_state, rules)
print(successors)  # Output: [{"integer", "real number"}, {"integer",
    "float"}]
```

```
Generated by Meta Llama 3.1-405

AutoToS prompt:

Provide only the Python function for the goal test function that tests
    , given a target fact, whether a state is a goal state.

Model response:
Python
```
```
def is_goal_state(state, target_fact):
    """
    Tests whether a state is a goal state given a target fact.

    Args:
        state (set): The current state of knowledge.
        target_fact: The target fact to be checked.

    Returns:
        bool: True if the state is a goal state, False otherwise.
    """
    return target_fact in state
```
```
Generated by Meta Llama 3.1-405

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Success: 4000, failure: 0, total expansions 88259, total evaluations
    528408
```

9.5 Sokoban

9.5.1 Deepseek-CoderV2

Listing 56: Without partial successor soundness test

```
AutoToS prompt:

This domain models the classic Sokoban game, where the robot needs to
    move around and push stones to specific locations while adhering
    to the constraints defined by the walls in the maze.
The maze is defined by a grid of values 0,1, and 2, where 2 means it
    is a goal location for a stone, 1 means the cell is blocked, and
    either 0 or 2 means that the cell can be occupied. A cell is clear
     if it can be occupied, but is not occupied by either the player
    or any stone.

The actions are:

move: This action moves the player in a specified direction. To
    perform this action, the target location must be clear and
    adjacent to player's location. Performing this action will result
    in the player being at the target location and no longer at the
    starting location, meaning the starting location is now clear,
    while the target location is now not clear.

push: This action allows the player to push a stone in a specified
    direction. To perform this action, the player's location, the
    stone location, and the target location must line up in the same
```

```
    direction, and the target location must be clear. Performing the
    action results in player being at the stone's initial location and
     the stone is at the target location. Further, the player's
    previous location is clear, while the target location is not clear
    .

Assume that states are defined by a dictionary with keys at-player and
     at-stone.
Here is an example of a state
{'at-player': (4, 4), 'at-stone': [(2, 2), (3, 3)]}
Here is an example of the grid:
[[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1,
    1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0,
    1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0,
     0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]

Provide a Python implementation of successor states function,
    receiving a state and a grid and returning a list of immediate
    successor states.

Model response:
python
```

```
def get_successor_states(state, grid):
    def is_clear(location):
        x, y = location
        return grid[x][y] == 0 or grid[x][y] == 2

    def is_valid_move(location):
        x, y = location
        return 0 <= x < len(grid) and 0 <= y < len(grid[0]) and
            is_clear(location)

    def move_player(direction):
        x, y = state['at-player']
        if direction == 'up':
            new_location = (x - 1, y)
        elif direction == 'down':
            new_location = (x + 1, y)
        elif direction == 'left':
            new_location = (x, y - 1)
        elif direction == 'right':
            new_location = (x, y + 1)
        else:
            return None

        if is_valid_move(new_location):
            new_state = state.copy()
            new_state['at-player'] = new_location
            return new_state
        return None

    def push_stone(direction):
        x, y = state['at-player']
        if direction == 'up':
            stone_location = (x - 1, y)
            new_stone_location = (x - 2, y)
        elif direction == 'down':
            stone_location = (x + 1, y)
            new_stone_location = (x + 2, y)
        elif direction == 'left':
            stone_location = (x, y - 1)
            new_stone_location = (x, y - 2)
        elif direction == 'right':
            stone_location = (x, y + 1)
```

```
            new_stone_location = (x, y + 2)
        else:
            return None

        if stone_location in state['at-stone'] and is_valid_move(
            new_stone_location):
            new_state = state.copy()
            new_state['at-player'] = stone_location
            new_state['at-stone'] = [new_stone_location if s ==
                stone_location else s for s in new_state['at-stone']]
            return new_state
        return None

    directions = ['up', 'down', 'left', 'right']
    successor_states = []

    for direction in directions:
        move_result = move_player(direction)
        if move_result:
            successor_states.append(move_result)

        push_result = push_stone(direction)
        if push_result:
            successor_states.append(push_result)

    return successor_states
``
```

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state for a given goal configuration. Assume that the goal configuration is provided by the 2-dimensional array as before. The function should receive a state and the 2-dimensional array and return True if the state is a goal state and False otherwise.

Model response:
python

```
def is_goal_state(state, grid):
    for stone in state['at-stone']:
        x, y = stone
        if grid[x][y] != 2:
            return False
    return True
``
```

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Found solution of length 19, optimal: 49
Found solution of length 9, optimal: 35
Found solution of length 3, optimal: 51
Found solution of length 27, optimal: 107
Found solution of length 1, optimal: 35
Found solution of length 21, optimal: 69
Found solution of length 17, optimal: 95
Found solution of length 16, optimal: 76
Found solution of length 13, optimal: 83
Found solution of length 15, optimal: 92

```
Found solution of length 7, optimal: 429
Total success: 0, failure: 11, total expansions 133465, total
    generated 421188
```

Listing 57: With partial successor soundness test

```
AutoToS prompt:

This domain models the classic Sokoban game, where the robot needs to
    move around and push stones to specific locations while adhering
    to the constraints defined by the walls in the maze.
The maze is defined by a grid of values 0,1, and 2, where 2 means it
    is a goal location for a stone, 1 means the cell is blocked, and
    either 0 or 2 means that the cell can be occupied. A cell is clear
     if it can be occupied, but is not occupied by either the player
    or any stone.

The actions are:

move: This action moves the player in a specified direction. To
    perform this action, the target location must be clear and
    adjacent to player's location. Performing this action will result
    in the player being at the target location and no longer at the
    starting location, meaning the starting location is now clear,
    while the target location is now not clear.

push: This action allows the player to push a stone in a specified
    direction. To perform this action, the player's location, the
    stone location, and the target location must line up in the same
    direction, and the target location must be clear. Performing the
    action results in player being at the stone's initial location and
     the stone is at the target location. Further, the player's
    previous location is clear, while the target location is not clear
    .

Assume that states are defined by a dictionary with keys at-player and
     at-stone.
Here is an example of a state
{'at-player': (4, 4), 'at-stone': [(2, 2), (3, 3)]}
Here is an example of the grid:
[[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1,
    1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0,
    1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0,
     0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]

Provide a Python implementation of successor states function,
    receiving a state and a grid and returning a list of immediate
    successor states.

Model response:
To implement the successor states function for the Sokoban game, we
    need to consider both the `move` and `push` actions. The function
    will take the current state and the grid as inputs and return a
    list of possible successor states based on the allowed actions.

Here's the Python implementation:

python
'''
def successor_states(state, grid):
    def is_clear(x, y):
        return 0 <= x < len(grid) and 0 <= y < len(grid[0]) and grid[x
            ][y] in [0, 2] and (x, y) not in state['at-stone']

    def move_player(dx, dy):
```

```
        x, y = state['at-player']
        new_x, new_y = x + dx, y + dy
        if is_clear(new_x, new_y):
            new_state = state.copy()
            new_state['at-player'] = (new_x, new_y)
            return new_state
        return None

    def push_stone(dx, dy):
        x, y = state['at-player']
        new_x, new_y = x + dx, y + dy
        if (new_x, new_y) in state['at-stone']:
            stone_new_x, stone_new_y = new_x + dx, new_y + dy
            if is_clear(stone_new_x, stone_new_y):
                new_state = state.copy()
                new_state['at-player'] = (new_x, new_y)
                new_state['at-stone'] = [(sx, sy) if (sx, sy) != (
                    new_x, new_y) else (stone_new_x, stone_new_y) for
                    sx, sy in new_state['at-stone']]
                return new_state
        return None

    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down,
        Left, Up
    successors = []

    for dx, dy in directions:
        # Try to move the player
        new_state = move_player(dx, dy)
        if new_state:
            successors.append(new_state)

        # Try to push a stone
        new_state = push_stone(dx, dy)
        if new_state:
            successors.append(new_state)

    return successors

# Example usage:
state = {'at-player': (4, 4), 'at-stone': [(2, 2), (3, 3)]}
grid = [
    [1, 1, 1, 1, 0, 0, 0, 0],
    [1, 0, 0, 1, 1, 0, 0, 0],
    [1, 0, 0, 0, 1, 1, 1, 1],
    [1, 0, 0, 0, 1, 0, 0, 1],
    [1, 1, 0, 0, 0, 0, 0, 1],
    [0, 1, 1, 1, 2, 2, 0, 1],
    [0, 0, 0, 1, 0, 1, 0, 1],
    [0, 0, 0, 1, 0, 0, 0, 1],
    [0, 0, 0, 1, 1, 1, 1, 1]
]

successors = successor_states(state, grid)
for s in successors:
    print(s)
```

This function defines two helper functions, `move_player` and `push_stone`, to handle the `move` and `push` actions, respectively. It then iterates over the possible directions and attempts to perform each action, collecting valid successor states in the `successors` list, which is returned at the end

AutoToS prompt:

````
Provide only the python code that tests whether a state is a goal
    state for a given goal configuration. Assume that the goal
    configuration is provided by the 2-dimensional array as before.
    The function should receive a state and the 2-dimensional array
    and return True if the state is a goal state and False otherwise.

Model response:
python
```
def is_goal_state(state, grid):
    for stone in state['at-stone']:
        if grid[stone[0]][stone[1]] != 2:
            return False
    return True

# Example usage:
state = {'at-player': (4, 4), 'at-stone': [(5, 4), (5, 5)]}
grid = [
    [1, 1, 1, 1, 0, 0, 0, 0],
    [1, 0, 0, 1, 1, 0, 0, 0],
    [1, 0, 0, 0, 1, 1, 1, 1],
    [1, 0, 0, 0, 1, 0, 0, 1],
    [1, 1, 0, 0, 0, 0, 0, 1],
    [0, 1, 1, 1, 2, 2, 0, 1],
    [0, 0, 0, 1, 0, 1, 0, 1],
    [0, 0, 0, 1, 0, 0, 0, 1],
    [0, 0, 0, 1, 1, 1, 1, 1]
]

print(is_goal_state(state, grid))  # Should return True
``

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Total success: 11, failure: 0, total expansions 763076, total
    generated 1953949
````

9.5.2 GPT4-Mini

Listing 58: Without partial successor soundness test

```
AutoToS prompt:

This domain models the classic Sokoban game, where the robot needs to
    move around and push stones to specific locations while adhering
    to the constraints defined by the walls in the maze.
The maze is defined by a grid of values 0,1, and 2, where 2 means it
    is a goal location for a stone, 1 means the cell is blocked, and
    either 0 or 2 means that the cell can be occupied. A cell is clear
     if it can be occupied, but is not occupied by either the player
    or any stone.

The actions are:

move: This action moves the player in a specified direction. To
    perform this action, the target location must be clear and
    adjacent to player's location. Performing this action will result
    in the player being at the target location and no longer at the
```
```

starting location, meaning the starting location is now clear, while the target location is now not clear.

push: This action allows the player to push a stone in a specified direction. To perform this action, the player's location, the stone location, and the target location must line up in the same direction, and the target location must be clear. Performing the action results in player being at the stone's initial location and the stone is at the target location. Further, the player's previous location is clear, while the target location is not clear.

Assume that states are defined by a dictionary with keys at-player and at-stone.
Here is an example of a state
{'at-player': (4, 4), 'at-stone': [(2, 2), (3, 3)]}
Here is an example of the grid:
[[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1, 1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0, 1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]

Provide a Python implementation of successor states function, receiving a state and a grid and returning a list of immediate successor states.

Model response:
```python
def successor_states(state, grid):
    def is_clear(x, y):
        return 0 <= x < len(grid) and 0 <= y < len(grid[0]) and (grid[
            x][y] == 0 or grid[x][y] == 2)

    def move_player(x, y, new_x, new_y):
        return (new_x, new_y), state['at-stone']

    def push_stone(player_x, player_y, stone_x, stone_y, new_x, new_y)
        :
        stones = state['at-stone'][:]
        stones.remove((stone_x, stone_y))
        stones.append((new_x, new_y))
        return (stone_x, stone_y), stones

    player_x, player_y = state['at-player']
    stone_positions = state['at-stone']
    successor_states = []

    directions = {
        "up": (-1, 0),
        "down": (1, 0),
        "left": (0, -1),
        "right": (0, 1)
    }

    for direction, (dx, dy) in directions.items():
        new_player_x = player_x + dx
        new_player_y = player_y + dy

        if is_clear(new_player_x, new_player_y):
            successor_states.append(move_player(player_x, player_y,
                new_player_x, new_player_y))

        for stone_x, stone_y in stone_positions:
            if (stone_x, stone_y) == (new_player_x, new_player_y):
                target_x = stone_x + dx
```

````
                target_y = stone_y + dy
                if is_clear(target_x, target_y):
                    successor_states.append(push_stone(player_x,
                        player_y, stone_x, stone_y, target_x, target_y
                        ))

    return successor_states
```

AutoToS prompt:

Provide only the python code that tests whether a state is a goal
    state for a given goal configuration. Assume that the goal
    configuration is provided by the 2-dimensional array as before.
    The function should receive a state and the 2-dimensional array
    and return True if the state is a goal state and False otherwise.

Model response:
```python
def is_goal_state(state, goal_configuration):
    def is_goal_position(pos):
        return goal_configuration[pos[0]][pos[1]] == 2

    stone_positions = state['at-stone']

    return all(is_goal_position(stone) for stone in stone_positions)
```

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
The following exception was produced when testing the successor
    function:
  File "<string>", line 5, in is_goal_state

tuple indices must be integers or slices, not str. Please fix this
    exception in the successor function.
Model response:
```python
def successor_states(state, grid):
    def is_clear(x, y):
        return 0 <= x < len(grid) and 0 <= y < len(grid[0]) and (grid[
            x][y] == 0 or grid[x][y] == 2)

    def move_player(x, y, new_x, new_y):
        return {'at-player': (new_x, new_y), 'at-stone': state['at-
            stone']}

    def push_stone(player_x, player_y, stone_x, stone_y, new_x, new_y)
        :
        stones = state['at-stone'][:]
        stones.remove((stone_x, stone_y))
        stones.append((new_x, new_y))
        return {'at-player': (stone_x, stone_y), 'at-stone': stones}

    player_x, player_y = state['at-player']
    stone_positions = state['at-stone']
    successor_states = []

    directions = {
        "up": (-1, 0),
        "down": (1, 0),
````

```
        "left": (0, -1),
        "right": (0, 1)
    }

    for direction, (dx, dy) in directions.items():
        new_player_x = player_x + dx
        new_player_y = player_y + dy

        if is_clear(new_player_x, new_player_y):
            successor_states.append(move_player(player_x, player_y,
                new_player_x, new_player_y))

        for stone_x, stone_y in stone_positions:
            if (stone_x, stone_y) == (new_player_x, new_player_y):
                target_x = stone_x + dx
                target_y = stone_y + dy
                if is_clear(target_x, target_y):
                    successor_states.append(push_stone(player_x,
                        player_y, stone_x, stone_y, target_x, target_y
                        ))

    return successor_states
```

```
System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Found solution of length 25, optimal: 69
Found solution of length 1, optimal: 35
Found solution of length 21, optimal: 95
Found solution of length 17, optimal: 92
Found solution of length 27, optimal: 49
Found solution of length 7, optimal: 429
Found solution of length 3, optimal: 51
Found solution of length 21, optimal: 83
Found solution of length 17, optimal: 35
Found solution of length 16, optimal: 76
Found solution of length 57, optimal: 107
Total success: 0, failure: 11, total expansions 364051, total
    generated 1109339
```

Listing 59: With partial successor soundness test

```
AutoToS prompt:

This domain models the classic Sokoban game, where the robot needs to
    move around and push stones to specific locations while adhering
    to the constraints defined by the walls in the maze.
The maze is defined by a grid of values 0,1, and 2, where 2 means it
    is a goal location for a stone, 1 means the cell is blocked, and
    either 0 or 2 means that the cell can be occupied. A cell is clear
     if it can be occupied, but is not occupied by either the player
    or any stone.

The actions are:

move: This action moves the player in a specified direction. To
    perform this action, the target location must be clear and
    adjacent to player’s location. Performing this action will result
    in the player being at the target location and no longer at the
    starting location, meaning the starting location is now clear,
    while the target location is now not clear.
```

push: This action allows the player to push a stone in a specified direction. To perform this action, the player's location, the stone location, and the target location must line up in the same direction, and the target location must be clear. Performing the action results in player being at the stone's initial location and the stone is at the target location. Further, the player's previous location is clear, while the target location is not clear.

Assume that states are defined by a dictionary with keys at-player and at-stone.
Here is an example of a state
{'at-player': (4, 4), 'at-stone': [(2, 2), (3, 3)]}
Here is an example of the grid:
[[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1, 1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0, 1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]

Provide a Python implementation of successor states function, receiving a state and a grid and returning a list of immediate successor states.

Model response:
```python
def successor_states(state, grid):
    def is_clear(position):
        x, y = position
        return grid[x][y] in (0, 2)

    def move_player(from_pos, to_pos):
        return {
            'at-player': to_pos,
            'at-stone': state['at-stone']
        }

    def push_stone(player_pos, stone_pos, target_pos):
        return {
            'at-player': stone_pos,
            'at-stone': [target_pos if stone == stone_pos else stone
                for stone in state['at-stone']]
        }

    x, y = state['at-player']
    stones = state['at-stone']
    directions = {
        'up': (-1, 0),
        'down': (1, 0),
        'left': (0, -1),
        'right': (0, 1)
    }

    successor_states = []

    for direction, (dx, dy) in directions.items():
        new_player_pos = (x + dx, y + dy)

        if 0 <= new_player_pos[0] < len(grid) and 0 <= new_player_pos
            [1] < len(grid[0]):
            if is_clear(new_player_pos):
                successor_states.append(move_player((x, y),
                    new_player_pos))

            for i, stone_pos in enumerate(stones):
                if stone_pos == new_player_pos:
```

```
                    target_pos = (stone_pos[0] + dx, stone_pos[1] + dy
                        )
                    if 0 <= target_pos[0] < len(grid) and 0 <=
                        target_pos[1] < len(grid[0]):
                        if is_clear(target_pos):
                            successor_states.append(push_stone((x, y),
                                stone_pos, target_pos))

    return successor_states
```

AutoToS prompt:

    Provide only the python code that tests whether a state is a goal
        state for a given goal configuration. Assume that the goal
        configuration is provided by the 2-dimensional array as before
        . The function should receive a state and the 2-dimensional
        array and return True if the state is a goal state and False
        otherwise.

Model response:
```python
def is_goal_state(state, goal_grid):
    def is_at_goal(stone_pos):
        x, y = stone_pos
        return goal_grid[x][y] == 2

    return all(is_at_goal(stone) for stone in state['at-stone'])
```

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
Error occurred for grid configuration: [[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1, 1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0, 1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]
Invalid transition: a stone and the player are at the same location.
Let’s think step by step. First think through in words why the successor function produced a successor that has a stone and the player at the same location. Then provide the complete Python code for the revised successor function that ensures that in all successors the player and the stones are at different locations.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: {’at-player’: (4, 3), ’at-stone’: [(2, 2), (3, 3)]}
Example wrong successor state: {’at-player’: (3, 3), ’at-stone’: [(2, 2), (3, 3)]}

Model response:
The issue arises when the successor function allows for the player to move directly onto a position occupied by a stone without checking whether the stone is already there. This results in invalid successor states where both the player and stone occupy the same location.

To resolve this, we need to ensure that when processing movements and pushes, we do not create any states where the player’s position coincides with the position of any stone. We can achieve this by adding a condition to check if the new player position is not already occupied by a stone before adding it to the list of successor states.

Here is the revised successor function:

```python
def successor_states(state, grid):
    def is_clear(position):
        x, y = position
        return grid[x][y] in (0, 2)

    def move_player(from_pos, to_pos):
        return {
            'at-player': to_pos,
            'at-stone': state['at-stone']
        }

    def push_stone(player_pos, stone_pos, target_pos):
        return {
            'at-player': stone_pos,
            'at-stone': [target_pos if stone == stone_pos else stone
                for stone in state['at-stone']]
        }

    x, y = state['at-player']
    stones = state['at-stone']
    directions = {
        'up': (-1, 0),
        'down': (1, 0),
        'left': (0, -1),
        'right': (0, 1)
    }

    successor_states = []

    for direction, (dx, dy) in directions.items():
        new_player_pos = (x + dx, y + dy)

        if 0 <= new_player_pos[0] < len(grid) and 0 <= new_player_pos
            [1] < len(grid[0]):
            if is_clear(new_player_pos) and new_player_pos not in
                stones:
                successor_states.append(move_player((x, y),
                    new_player_pos))

            for i, stone_pos in enumerate(stones):
                if stone_pos == new_player_pos:
                    target_pos = (stone_pos[0] + dx, stone_pos[1] + dy
                        )
                    if 0 <= target_pos[0] < len(grid) and 0 <=
                        target_pos[1] < len(grid[0]):
                        if is_clear(target_pos):
                            successor_states.append(push_stone((x, y),
                                stone_pos, target_pos))

    return successor_states
```

Successor Iteration 2

AutoToS prompt:
Error occurred for grid configuration: [[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1, 1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0, 1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]

Invalid transition: multiple stones at the same location.
Let's think step by step. First think through in words why the successor function produced a successor that has two stones at the same location. Then provide the complete Python code for the revised successor function that ensures that in all successors all stones are at different locations.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: {'at-player': (2, 1), 'at-stone': [(2, 2), (2, 3)]}
Example wrong successor state: {'at-player': (2, 2), 'at-stone': [(2, 3), (2, 3)]}

Model response:
The issue arises when we allow the same stone to be pushed to a target location that is already occupied by another stone. This can occur especially if the new position of the stone after the push coincides with an existing stone's position in the successor state.

To avoid this situation, we need to ensure that when we push a stone, the target position is clear of all other stones. We can do this by adding a check to see if the target position already contains a stone in the successor state before proceeding with any push.

Here is the revised successor function with the necessary checks:

```python
def successor_states(state, grid):
    def is_clear(position):
        x, y = position
        return grid[x][y] in (0, 2)

    def move_player(from_pos, to_pos):
        return {
            'at-player': to_pos,
            'at-stone': state['at-stone']
        }

    def push_stone(player_pos, stone_pos, target_pos):
        return {
            'at-player': stone_pos,
            'at-stone': [target_pos if stone == stone_pos else stone
                for stone in state['at-stone']]
        }

    x, y = state['at-player']
    stones = state['at-stone']
    directions = {
        'up': (-1, 0),
        'down': (1, 0),
        'left': (0, -1),
        'right': (0, 1)
    }

    successor_states = []

    for direction, (dx, dy) in directions.items():
        new_player_pos = (x + dx, y + dy)

        if 0 <= new_player_pos[0] < len(grid) and 0 <= new_player_pos
            [1] < len(grid[0]):
            if is_clear(new_player_pos) and new_player_pos not in
                stones:
```

```
                successor_states.append(move_player((x, y),
                    new_player_pos))

            for stone_pos in stones:
                if stone_pos == new_player_pos:
                    target_pos = (stone_pos[0] + dx, stone_pos[1] + dy
                        )
                    if 0 <= target_pos[0] < len(grid) and 0 <=
                        target_pos[1] < len(grid[0]):
                        if is_clear(target_pos) and target_pos not in
                            stones:
                            successor_states.append(push_stone((x, y),
                                 stone_pos, target_pos))

    return successor_states
```

```
System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Total success: 11, failure: 0, total expansions 764128, total
    generated 1956111
```

9.5.3 GPT4

Listing 60: Without partial successor soundness test

```
AutoToS prompt:

This domain models the classic Sokoban game, where the robot needs to
    move around and push stones to specific locations while adhering
    to the constraints defined by the walls in the maze.
The maze is defined by a grid of values 0,1, and 2, where 2 means it
    is a goal location for a stone, 1 means the cell is blocked, and
    either 0 or 2 means that the cell can be occupied. A cell is clear
     if it can be occupied, but is not occupied by either the player
    or any stone.

The actions are:

move: This action moves the player in a specified direction. To
    perform this action, the target location must be clear and
    adjacent to player’s location. Performing this action will result
    in the player being at the target location and no longer at the
    starting location, meaning the starting location is now clear,
    while the target location is now not clear.

push: This action allows the player to push a stone in a specified
    direction. To perform this action, the player’s location, the
    stone location, and the target location must line up in the same
    direction, and the target location must be clear. Performing the
    action results in player being at the stone’s initial location and
     the stone is at the target location. Further, the player’s
    previous location is clear, while the target location is not clear
    .

Assume that states are defined by a dictionary with keys at-player and
     at-stone.
Here is an example of a state
{’at-player’: (4, 4), ’at-stone’: [(2, 2), (3, 3)]}
Here is an example of the grid:
[[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1,
    1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0,
```

1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]

Provide a Python implementation of successor states function, receiving a state and a grid and returning a list of immediate successor states.

Model response:

```python
def sokoban_successor_states(state, grid):
    def is_clear(cell, state, grid):
        """Check if the cell is clear and therefore can be occupied"""
        r, c = cell
        return grid[r][c] != 1 and cell != state['at-player'] and cell
             not in state['at-stone']

    def move(player_pos, direction, state, grid):
        """Move the player in the specified direction if the target
            cell is clear"""
        r, c = player_pos
        dr, dc = direction
        new_pos = (r + dr, c + dc)
        if is_clear(new_pos, state, grid):
            new_state = state.copy()
            new_state['at-player'] = new_pos
            return new_state
        return None

    def push(player_pos, direction, state, grid):
        """Push a stone if the conditions are met"""
        r, c = player_pos
        dr, dc = direction
        stone_pos = (r + dr, c + dc)
        target_pos = (r + 2 * dr, c + 2 * dc)
        if stone_pos in state['at-stone'] and is_clear(target_pos,
            state, grid):
            new_state = {
                'at-player': stone_pos,
                'at-stone': [target_pos if pos == stone_pos else pos
                    for pos in state['at-stone']]
            }
            return new_state
        return None

    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    player_pos = state['at-player']

    successors = []

    for direction in directions:
        # Try moving the player
        new_state = move(player_pos, direction, state, grid)
        if new_state is not None:
            successors.append(new_state)

        # Try pushing a stone
        new_state = push(player_pos, direction, state, grid)
        if new_state is not None:
            successors.append(new_state)

    return successors
```

AutoToS prompt:

```
Provide only the python code that tests whether a state is a goal
    state for a given goal configuration. Assume that the goal
    configuration is provided by the 2-dimensional array as before.
    The function should receive a state and the 2-dimensional array
    and return True if the state is a goal state and False otherwise.

Model response:
```python
def is_goal_state(state, goal_grid):
    def is_goal_position(position, goal_grid):
        r, c = position
        return goal_grid[r][c] == 2

    for stone in state['at-stone']:
        if not is_goal_position(stone, goal_grid):
            return False

    return True
```

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Total success: 11, failure: 0, total expansions 764128, total
    generated 1956111
```

Listing 61: With partial successor soundness test

```
AutoToS prompt:

This domain models the classic Sokoban game, where the robot needs to
    move around and push stones to specific locations while adhering
    to the constraints defined by the walls in the maze.
The maze is defined by a grid of values 0,1, and 2, where 2 means it
    is a goal location for a stone, 1 means the cell is blocked, and
    either 0 or 2 means that the cell can be occupied. A cell is clear
     if it can be occupied, but is not occupied by either the player
    or any stone.

The actions are:

move: This action moves the player in a specified direction. To
    perform this action, the target location must be clear and
    adjacent to player's location. Performing this action will result
    in the player being at the target location and no longer at the
    starting location, meaning the starting location is now clear,
    while the target location is now not clear.

push: This action allows the player to push a stone in a specified
    direction. To perform this action, the player's location, the
    stone location, and the target location must line up in the same
    direction, and the target location must be clear. Performing the
    action results in player being at the stone's initial location and
     the stone is at the target location. Further, the player's
    previous location is clear, while the target location is not clear
    .

Assume that states are defined by a dictionary with keys at-player and
     at-stone.
Here is an example of a state
```

{'at-player': (4, 4), 'at-stone': [(2, 2), (3, 3)]}
Here is an example of the grid:
[[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1, 1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0, 1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]

Provide a Python implementation of successor states function, receiving a state and a grid and returning a list of immediate successor states.

Model response:

```python
def successor_states(state, grid):
    def is_clear(location):
        r, c = location
        return (0 <= r < len(grid) and 0 <= c < len(grid[0]) and
                grid[r][c] != 1 and
                location not in state['at-stone'] and
                location != state['at-player'])

    def move_player(current_state, direction):
        deltas = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), '
            right': (0, 1)}
        if direction in deltas:
            dr, dc = deltas[direction]
            r, c = current_state['at-player']
            new_location = (r + dr, c + dc)
            if is_clear(new_location):
                new_state = current_state.copy()
                new_state['at-player'] = new_location
                return new_state
        return None

    def push_stone(current_state, direction):
        deltas = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), '
            right': (0, 1)}
        if direction in deltas:
            dr, dc = deltas[direction]
            r, c = current_state['at-player']
            stone_location = (r + dr, c + dc)
            if stone_location in current_state['at-stone']:
                target_location = (stone_location[0] + dr,
                    stone_location[1] + dc)
                if is_clear(target_location):
                    new_state = current_state.copy()
                    new_state['at-player'] = stone_location
                    new_state['at-stone'].remove(stone_location)
                    new_state['at-stone'].append(target_location)
                    return new_state
        return None

    successor_states = []
    directions = ['up', 'down', 'left', 'right']

    for direction in directions:
        new_state = move_player(state, direction)
        if new_state:
            successor_states.append(new_state)

        new_state = push_stone(state, direction)
        if new_state:
            successor_states.append(new_state)

    return successor_states
```

```
AutoToS prompt:

    Provide only the python code that tests whether a state is a goal
        state for a given goal configuration. Assume that the goal
        configuration is provided by the 2-dimensional array as before
        . The function should receive a state and the 2-dimensional
        array and return True if the state is a goal state and False
        otherwise.

Model response:
```python
def is_goal_state(state, goal_grid):
    goal_positions = set()
    for r in range(len(goal_grid)):
        for c in range(len(goal_grid[0])):
            if goal_grid[r][c] == 2:
                goal_positions.add((r, c))

    return set(state['at-stone']) == goal_positions
```

System message:
Goal Soundness Test Passed
Soundness test
Successor Iteration 1

AutoToS prompt:
Error occurred for grid configuration: [[1, 1, 1, 1, 0, 0, 0, 0], [1,
    0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1, 1, 1, 1], [1, 0, 0, 0, 1, 0,
     0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0, 1, 1, 1, 2, 2, 0, 1], [0, 0,
     0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 1,
    1, 1]]
Input state should not change as a result of applying the successor
    function.
State before successor function applied: {'at-player': (4, 3), 'at-
    stone': [(2, 2), (3, 3)]}
State after successor function applied: {'at-player': (4, 3), 'at-
    stone': [(2, 2), (2, 3)]}
Let's think step by step. First, think of the possible reasons why the
     input state may change as a result of applying the successor
    function, such as the use of shallow copy.
Then, provide the complete Python code for the revised successor
    function that returns a list of successor states and ensure it
    does not change the input state.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.

Model response:
```python
def successor_states(state, grid):
    import copy

    def is_clear(location):
        r, c = location
        return (0 <= r < len(grid) and 0 <= c < len(grid[0]) and
                grid[r][c] != 1 and
                location not in state['at-stone'] and
                location != state['at-player'])

    def move_player(current_state, direction):
        deltas = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), '
            right': (0, 1)}
        if direction in deltas:
```
```

```
            dr, dc = deltas[direction]
            r, c = current_state['at-player']
            new_location = (r + dr, c + dc)
            if is_clear(new_location):
                new_state = copy.deepcopy(current_state)
                new_state['at-player'] = new_location
                return new_state
        return None

    def push_stone(current_state, direction):
        deltas = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), '
            right': (0, 1)}
        if direction in deltas:
            dr, dc = deltas[direction]
            r, c = current_state['at-player']
            stone_location = (r + dr, c + dc)
            if stone_location in current_state['at-stone']:
                target_location = (stone_location[0] + dr,
                    stone_location[1] + dc)
                if is_clear(target_location):
                    new_state = copy.deepcopy(current_state)
                    new_state['at-player'] = stone_location
                    new_state['at-stone'].remove(stone_location)
                    new_state['at-stone'].append(target_location)
                    return new_state
        return None

    successor_states = []
    directions = ['up', 'down', 'left', 'right']

    for direction in directions:
        new_state = move_player(state, direction)
        if new_state:
            successor_states.append(new_state)

        new_state = push_stone(state, direction)
        if new_state:
            successor_states.append(new_state)

    return successor_states
```

```
System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Total success: 11, failure: 0, total expansions 764128, total
    generated 1956111
```

9.5.4 Llama3.1 70B

Listing 62: Without partial successor soundness test

```
AutoToS prompt:

This domain models the classic Sokoban game, where the robot needs to
    move around and push stones to specific locations while adhering
    to the constraints defined by the walls in the maze.
The maze is defined by a grid of values 0,1, and 2, where 2 means it
    is a goal location for a stone, 1 means the cell is blocked, and
    either 0 or 2 means that the cell can be occupied. A cell is clear
     if it can be occupied, but is not occupied by either the player
    or any stone.

The actions are:
```

```
move: This action moves the player in a specified direction. To
    perform this action, the target location must be clear and
    adjacent to player’s location. Performing this action will result
    in the player being at the target location and no longer at the
    starting location, meaning the starting location is now clear,
    while the target location is now not clear.

push: This action allows the player to push a stone in a specified
    direction. To perform this action, the player’s location, the
    stone location, and the target location must line up in the same
    direction, and the target location must be clear. Performing the
    action results in player being at the stone’s initial location and
     the stone is at the target location. Further, the player’s
    previous location is clear, while the target location is not clear
    .

Assume that states are defined by a dictionary with keys at-player and
     at-stone.
Here is an example of a state
{’at-player’: (4, 4), ’at-stone’: [(2, 2), (3, 3)]}
Here is an example of the grid:
[[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1,
    1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0,
    1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0,
     0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]

Provide a Python implementation of successor states function,
    receiving a state and a grid and returning a list of immediate
    successor states.

Model response:
Python
```

```
import copy

def successor_states(state, grid):
    def is_clear(location):
        return grid[location[0]][location[1]] in [0, 2]

    def adjacent_locations(location, direction):
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        return (location[0] + directions[direction][0], location[1] +
            directions[direction][1])

    def move(location, direction):
        new_location = adjacent_locations(location, direction)
        if is_clear(new_location):
            return {’at-player’: new_location, ’at-stone’: state[’at-
                stone’]}

    def push(location, direction):
        stone_location = adjacent_locations(location, direction)
        if stone_location in state[’at-stone’]:
            target_location = adjacent_locations(stone_location,
                direction)
            if is_clear(target_location):
                new_stone_locations = state[’at-stone’].copy()
                new_stone_locations.remove(stone_location)
                new_stone_locations.append(target_location)
                return {’at-player’: stone_location, ’at-stone’:
                    new_stone_locations}

    player_location = state[’at-player’]
    successor_states = []
```

````
    for direction in range(4):
        new_state = move(player_location, direction)
        if new_state:
            successor_states.append(new_state)
        new_state = push(player_location, direction)
        if new_state:
            successor_states.append(new_state)

    return successor_states
``

AutoToS prompt:

Provide only the python code that tests whether a state is a goal
    state for a given goal configuration. Assume that the goal
    configuration is provided by the 2-dimensional array as before.
    The function should receive a state and the 2-dimensional array
    and return True if the state is a goal state and False otherwise.

Model response:
Python
```
def is_goal_state(state, grid):
    goal_locations = [(i, j) for i in range(len(grid)) for j in range(
        len(grid[0])) if grid[i][j] == 2]
    return sorted(state['at-stone']) == sorted(goal_locations)
``

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Found solution of length 7, optimal: 429
Found solution of length 26, optimal: 95
Found solution of length 3, optimal: 51
Found solution of length 17, optimal: 35
Found solution of length 16, optimal: 76
Found solution of length 17, optimal: 92
Found solution of length 23, optimal: 83
Found solution of length 25, optimal: 69
Found solution of length 30, optimal: 49
Found solution of length 3, optimal: 35
Found solution of length 61, optimal: 107
Total success: 0, failure: 11, total expansions 519757, total
    generated 1553561
````

Listing 63: With partial successor soundness test

````
AutoToS prompt:

This domain models the classic Sokoban game, where the robot needs to
    move around and push stones to specific locations while adhering
    to the constraints defined by the walls in the maze.
The maze is defined by a grid of values 0,1, and 2, where 2 means it
    is a goal location for a stone, 1 means the cell is blocked, and
    either 0 or 2 means that the cell can be occupied. A cell is clear
     if it can be occupied, but is not occupied by either the player
    or any stone.

The actions are:
````
````

```
move: This action moves the player in a specified direction. To
    perform this action, the target location must be clear and
    adjacent to player's location. Performing this action will result
    in the player being at the target location and no longer at the
    starting location, meaning the starting location is now clear,
    while the target location is now not clear.

push: This action allows the player to push a stone in a specified
    direction. To perform this action, the player's location, the
    stone location, and the target location must line up in the same
    direction, and the target location must be clear. Performing the
    action results in player being at the stone's initial location and
     the stone is at the target location. Further, the player's
    previous location is clear, while the target location is not clear
    .

Assume that states are defined by a dictionary with keys at-player and
     at-stone.
Here is an example of a state
{'at-player': (4, 4), 'at-stone': [(2, 2), (3, 3)]}
Here is an example of the grid:
[[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1,
    1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0,
    1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0,
     0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]

Provide a Python implementation of successor states function,
    receiving a state and a grid and returning a list of immediate
    successor states.

Model response:
Python
```

```
import copy

def successor_states(state, grid):
    def is_clear(location):
        return grid[location[0]][location[1]] != 1 and (location not
            in state['at-stone'] and location != state['at-player'])

    def move(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        if is_clear(new_player_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            return new_state

    def push(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        new_stone_location = (new_player_location[0] + direction[0],
            new_player_location[1] + direction[1])
        if new_stone_location in state['at-stone'] and is_clear(
            new_stone_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            new_state['at-stone'].remove(new_stone_location)
            new_state['at-stone'].append((new_stone_location[0] +
                direction[0], new_stone_location[1] + direction[1]))
            return new_state

    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    successors = []
```

````
    for direction in directions:
        move_successor = move(direction)
        if move_successor:
            successors.append(move_successor)
        push_successor = push(direction)
        if push_successor:
            successors.append(push_successor)
    return successors
``

AutoToS prompt:

Provide only the python code that tests whether a state is a goal
    state for a given goal configuration. Assume that the goal
    configuration is provided by the 2-dimensional array as before.
    The function should receive a state and the 2-dimensional array
    and return True if the state is a goal state and False otherwise.

Model response:
Python
```
def is_goal_state(state, grid):
    goal_locations = [(i, j) for i in range(len(grid)) for j in range(
        len(grid[0])) if grid[i][j] == 2]
    return sorted(state['at-stone']) == sorted(goal_locations)
``

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 1

AutoToS prompt:
Error occurred for grid configuration: [[1, 1, 1, 1, 1, 1], [1, 0, 0,
    2, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 2, 1,
    1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]]
Successor function when run on the state {'at-player': (5, 3), 'at-
    stone': [(5, 2), (4, 3)]} failed to produce all successors.
Missing successors are: [{'at-player': (5, 2), 'at-stone': [(5, 1),
    (4, 3)]}]
First think step by step why the successor function failed to produce
    all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.

Model response:
The successor function failed to produce all successors because it
    didn't consider the case where the player can push a stone to a
    goal location. In the given state, the player can push the stone
    at (5, 2) to the goal location (5, 1), but the successor function
    missed this case.
Here's the corrected successor function:
Python
```
import copy

def successor_states(state, grid):
    def is_clear(location):
        return grid[location[0]][location[1]] != 1 and (location not
            in state['at-stone'] and location != state['at-player'])
````

```
    def move(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        if is_clear(new_player_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            return new_state

    def push(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        new_stone_location = (new_player_location[0] + direction[0],
            new_player_location[1] + direction[1])
        if new_stone_location in state['at-stone'] and is_clear(
            new_stone_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            new_state['at-stone'].remove((new_stone_location[0] -
                direction[0], new_stone_location[1] - direction[1]))
            new_state['at-stone'].append(new_stone_location)
            return new_state

    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    successors = []
    for direction in directions:
        move_successor = move(direction)
        if move_successor:
            successors.append(move_successor)
        push_successor = push(direction)
        if push_successor:
            successors.append(push_successor)
    return successors
```

The fix was in the push function, where I changed new_state['at-stone'].append((new_stone_location[0] + direction[0], new_stone_location[1] + direction[1])) to new_state['at-stone'].append(new_stone_location). This correctly updates the stone's location after pushing it

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 2

AutoToS prompt:
Error occurred for grid configuration: [[1, 1, 1, 1, 1, 1], [1, 0, 0, 2, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 2, 1, 1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]]
Successor function when run on the state {'at-player': (5, 3), 'at-stone': [(5, 2), (4, 3)]} failed to produce all successors.
Missing successors are: [{'at-player': (5, 2), 'at-stone': [(5, 1), (4, 3)]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The successor function failed to produce all successors because it didn't consider the case where the player can push a stone to a new location that is not necessarily a goal location. In the given state, the player can push the stone at (5, 2) to the location (5, 1), but the successor function missed this case.

Here's the corrected successor function:
Python
```
import copy

def successor_states(state, grid):
    def is_clear(location):
        return grid[location[0]][location[1]] != 1 and (location not
            in state['at-stone'] and location != state['at-player'])

    def move(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        if is_clear(new_player_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            return new_state

    def push(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        new_stone_location = (new_player_location[0] + direction[0],
            new_player_location[1] + direction[1])
        if new_stone_location in state['at-stone'] and is_clear(
            new_stone_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            stone_index = state['at-stone'].index((new_stone_location
                [0] - direction[0], new_stone_location[1] - direction
                [1]))
            new_state['at-stone'][stone_index] = new_stone_location
            return new_state

    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    successors = []
    for direction in directions:
        move_successor = move(direction)
        if move_successor:
            successors.append(move_successor)
        push_successor = push(direction)
        if push_successor:
            successors.append(push_successor)
    return successors
```
The fix was in the push function, where I changed new_state['at-stone'].remove((new_stone_location[0] - direction[0], new_stone_location[1] - direction[1])) to stone_index = state['at-stone'].index((new_stone_location[0] - direction[0], new_stone_location[1] - direction[1])) and then new_state['at-stone'][stone_index] = new_stone_location. This correctly updates the stone's location after pushing it

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 3

AutoToS prompt:
Error occurred for grid configuration: [[1, 1, 1, 1, 1, 1], [1, 0, 0, 2, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 2, 1, 1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]]
Successor function when run on the state {'at-player': (5, 3), 'at-stone': [(5, 2), (4, 3)]} failed to produce all successors.
Missing successors are: [{'at-player': (5, 2), 'at-stone': [(5, 1), (4, 3)]}]

First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The successor function failed to produce all successors because it didn't consider the case where the player can push a stone to a new location that is not necessarily a goal location, and the stone is not at the edge of the grid. In the given state, the player can push the stone at (5, 2) to the location (5, 1), but the successor function missed this case.
Here's the corrected successor function:
Python
```
import copy

def successor_states(state, grid):
    def is_clear(location):
        return grid[location[0]][location[1]] != 1 and (location not
            in state['at-stone'] and location != state['at-player'])

    def move(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        if is_clear(new_player_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            return new_state

    def push(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        new_stone_location = (new_player_location[0] + direction[0],
            new_player_location[1] + direction[1])
        if new_stone_location in state['at-stone'] and is_clear(
            new_stone_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            stone_index = state['at-stone'].index(new_stone_location)
            new_state['at-stone'][stone_index] = (new_stone_location
                [0] - direction[0], new_stone_location[1] - direction
                [1])
            return new_state

    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    successors = []
    for direction in directions:
        move_successor = move(direction)
        if move_successor:
            successors.append(move_successor)
        push_successor = push(direction)
        if push_successor:
            successors.append(push_successor)
    return successors
```
The fix was in the push function, where I changed stone_index = state['at-stone'].index((new_stone_location[0] - direction[0], new_stone_location[1] - direction[1])) to stone_index = state['at-stone'].index(new_stone_location). This correctly updates the stone's location after pushing it

System message:
Successor States Soundness Test Passed

```
Completeness test
Successor Iteration 4

AutoToS prompt:
Error occurred for grid configuration: [[1, 1, 1, 1, 1, 1], [1, 0, 0,
    2, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 2, 1,
    1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]]
Successor function when run on the state {'at-player': (5, 3), 'at-
    stone': [(5, 2), (4, 3)]} failed to produce all successors.
Missing successors are: [{'at-player': (5, 2), 'at-stone': [(5, 1),
    (4, 3)]}]
First think step by step why the successor function failed to produce
    all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.

Model response:
The successor function failed to produce all successors because it
    didn't correctly handle the case where the player pushes a stone
    to a new location. In the given state, the player can push the
    stone at (5, 2) to the location (5, 1), but the successor function
     missed this case.
Here's the corrected successor function:
Python
```

```
import copy

def successor_states(state, grid):
    def is_clear(location):
        return grid[location[0]][location[1]] != 1 and (location not
            in state['at-stone'] and location != state['at-player'])

    def move(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        if is_clear(new_player_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            return new_state

    def push(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        new_stone_location = (new_player_location[0] + direction[0],
            new_player_location[1] + direction[1])
        if new_stone_location in state['at-stone'] and is_clear((
            new_stone_location[0] + direction[0], new_stone_location
            [1] + direction[1])):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            stone_index = state['at-stone'].index(new_stone_location)
            new_state['at-stone'][stone_index] = (new_stone_location
                [0] + direction[0], new_stone_location[1] + direction
                [1])
            return new_state

    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    successors = []
    for direction in directions:
        move_successor = move(direction)
        if move_successor:
            successors.append(move_successor)
        push_successor = push(direction)
        if push_successor:
```

```
            successors.append(push_successor)
    return successors
```

The fix was in the push function, where I changed the condition to check if the new stone location is clear, and updated the stone's location correctly

Successor Iteration 5

AutoToS prompt:

Error occurred for grid configuration: [[1, 1, 1, 1, 1, 1], [1, 0, 0, 2, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 2, 1, 1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]]

Invalid transition: a stone and the player are at the same location.

Let's think step by step. First think through in words why the successor function produced a successor that has a stone and the player at the same location. Then provide the complete Python code for the revised successor function that ensures that in all successors the player and the stones are at different locations.

Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: {'at-player': (5, 3), 'at-stone': [(3, 3), (4, 3)]}

Example wrong successor state: {'at-player': (4, 3), 'at-stone': [(2, 3), (4, 3)]}

Model response:

The successor function produced a successor with a stone and the player at the same location because it didn't check if the new player location is already occupied by a stone when pushing a stone.

Here's the revised successor function that ensures the player and stones are at different locations:

Python

```
import copy

def successor_states(state, grid):
    def is_clear(location):
        return grid[location[0]][location[1]] != 1 and (location not
            in state['at-stone'] and location != state['at-player'])

    def move(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        if is_clear(new_player_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            return new_state

    def push(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        new_stone_location = (new_player_location[0] + direction[0],
            new_player_location[1] + direction[1])
        if new_stone_location in state['at-stone'] and is_clear(
            new_stone_location) and (new_player_location[0] +
            direction[0], new_player_location[1] + direction[1]) not
            in state['at-stone']:
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            stone_index = state['at-stone'].index(new_stone_location)
            new_state['at-stone'][stone_index] = (new_stone_location
                [0] + direction[0], new_stone_location[1] + direction
                [1])
            return new_state
```

```
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    successors = []
    for direction in directions:
        move_successor = move(direction)
        if move_successor:
            successors.append(move_successor)
        push_successor = push(direction)
        if push_successor:
            successors.append(push_successor)
    return successors
```
The fix was in the push function, where I added an additional condition to check if the new player location is not already occupied by a stone

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 6

AutoToS prompt:
Error occurred for grid configuration: [[1, 1, 1, 1, 1, 1], [1, 0, 0, 2, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 2, 1, 1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]]
Successor function when run on the state {'at-player': (5, 3), 'at-stone': [(5, 2), (4, 3)]} failed to produce all successors.
Missing successors are: [{'at-player': (5, 2), 'at-stone': [(5, 1), (4, 3)]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
The successor function failed to produce all successors because it didn't correctly handle the case where the player pushes a stone to a new location. In the given state, the player can push the stone at (5, 2) to the location (5, 1), but the successor function missed this case.
Here's the corrected successor function:
Python
```
import copy

def successor_states(state, grid):
    def is_clear(location):
        return grid[location[0]][location[1]] != 1 and (location not
            in state['at-stone'] and location != state['at-player'])

    def move(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        if is_clear(new_player_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            return new_state

    def push(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        new_stone_location = (new_player_location[0] + direction[0],
            new_player_location[1] + direction[1])
```

```
        if new_stone_location in state['at-stone'] and is_clear((
            new_stone_location[0] + direction[0], new_stone_location
            [1] + direction[1])):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            stone_index = state['at-stone'].index(new_stone_location)
            new_state['at-stone'][stone_index] = (new_stone_location
                [0] + direction[0], new_stone_location[1] + direction
                [1])
            return new_state
        elif (new_player_location in state['at-stone'] and is_clear((
            new_player_location[0] + direction[0], new_player_location
            [1] + direction[1]))):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            stone_index = state['at-stone'].index(new_player_location)
            new_state['at-stone'][stone_index] = (new_player_location
                [0] + direction[0], new_player_location[1] + direction
                [1])
            return new_state

    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    successors = []
    for direction in directions:
        move_successor = move(direction)
        if move_successor:
            successors.append(move_successor)
        push_successor = push(direction)
        if push_successor:
            successors.append(push_successor)
    return successors
```

```
The fix was in the push function, where I added an additional
    condition to check if the player can push a stone to a new
    location when the player is already at the stone's location
Successor Iteration 7

AutoToS prompt:
Error occurred for grid configuration: [[1, 1, 1, 1, 1, 1], [1, 0, 0,
    2, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 2, 1,
    1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]]
Invalid transition: a stone and the player are at the same location.
Let's think step by step. First think through in words why the
    successor function produced a successor that has a stone and the
    player at the same location. Then provide the complete Python code
     for the revised successor function that ensures that in all
    successors the player and the stones are at different locations.
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.

Input state: {'at-player': (5, 3), 'at-stone': [(3, 3), (4, 3)]}
Example wrong successor state: {'at-player': (4, 3), 'at-stone': [(2,
    3), (4, 3)]}

Model response:
The successor function produced a successor with a stone and the
    player at the same location because it didn't check if the new
    player location is already occupied by a stone when pushing a
    stone.
Here's the revised successor function that ensures the player and
    stones are at different locations:
Python
```

```
import copy
```

```
def successor_states(state, grid):
    def is_clear(location):
        return grid[location[0]][location[1]] != 1 and (location not
            in state['at-stone'] and location != state['at-player'])

    def move(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        if is_clear(new_player_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            return new_state

    def push(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        new_stone_location = (new_player_location[0] + direction[0],
            new_player_location[1] + direction[1])
        if new_stone_location in state['at-stone'] and is_clear((
            new_stone_location[0] + direction[0], new_stone_location
            [1] + direction[1])) and (new_player_location[0] +
            direction[0], new_player_location[1] + direction[1]) not
            in state['at-stone']:
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            stone_index = state['at-stone'].index(new_stone_location)
            new_state['at-stone'][stone_index] = (new_stone_location
                [0] + direction[0], new_stone_location[1] + direction
                [1])
            return new_state

    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    successors = []
    for direction in directions:
        move_successor = move(direction)
        if move_successor:
            successors.append(move_successor)
        push_successor = push(direction)
        if push_successor and push_successor['at-player'] not in
            push_successor['at-stone']:
            successors.append(push_successor)
    return successors
```

The fix was in the push function, where I added an additional condition to check if the new player location is not already occupied by a stone. Also, I added a check before appending the push_successor to successors to ensure the player and stones are at different locations

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 8

AutoToS prompt:
Error occurred for grid configuration: [[1, 1, 1, 1, 1, 1], [1, 0, 0, 2, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 2, 1, 1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]]
Successor function when run on the state {'at-player': (5, 3), 'at-stone': [(5, 2), (4, 3)]} failed to produce all successors.
Missing successors are: [{'at-player': (5, 2), 'at-stone': [(5, 1), (4, 3)]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.

```
Remember how you fixed the previous mistakes, if any. Keep the same
    function signature.

Model response:
The successor function failed to produce all successors because it
    didn't correctly handle the case where the player pushes a stone
    to a new location. In the given state, the player can push the
    stone at (5, 2) to the location (5, 1), but the successor function
     missed this case.
Here's the corrected successor function:
Python
```
```
import copy

def successor_states(state, grid):
    def is_clear(location):
        return grid[location[0]][location[1]] != 1 and (location not
            in state['at-stone'] and location != state['at-player'])

    def move(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        if is_clear(new_player_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            return new_state

    def push(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        new_stone_location = (new_player_location[0] + direction[0],
            new_player_location[1] + direction[1])
        if new_stone_location in state['at-stone'] and is_clear((
            new_stone_location[0] + direction[0], new_stone_location
            [1] + direction[1])):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            stone_index = state['at-stone'].index(new_stone_location)
            new_state['at-stone'][stone_index] = (new_stone_location
                [0] + direction[0], new_stone_location[1] + direction
                [1])
            return new_state
        elif new_player_location in state['at-stone'] and is_clear((
            new_player_location[0] + direction[0], new_player_location
            [1] + direction[1])):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            stone_index = state['at-stone'].index(new_player_location)
            new_state['at-stone'][stone_index] = (new_player_location
                [0] + direction[0], new_player_location[1] + direction
                [1])
            return new_state

    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    successors = []
    for direction in directions:
        move_successor = move(direction)
        if move_successor:
            successors.append(move_successor)
        push_successor = push(direction)
        if push_successor:
            successors.append(push_successor)
    return successors
```

The fix was in the push function, where I added an additional condition to check if the player can push a stone to a new location when the player is already at the stone's location
Successor Iteration 9

AutoToS prompt:
Error occurred for grid configuration: [[1, 1, 1, 1, 1, 1], [1, 0, 0, 2, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 2, 1, 1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]]
Invalid transition: a stone and the player are at the same location.
Let's think step by step. First think through in words why the successor function produced a successor that has a stone and the player at the same location. Then provide the complete Python code for the revised successor function that ensures that in all successors the player and the stones are at different locations.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Input state: {'at-player': (5, 3), 'at-stone': [(3, 3), (4, 3)]}
Example wrong successor state: {'at-player': (4, 3), 'at-stone': [(2, 3), (4, 3)]}

Model response:
The successor function produced a successor with a stone and the player at the same location because it didn't check if the new player location is already occupied by a stone when pushing a stone.
Here's the revised successor function that ensures the player and stones are at different locations:
Python
```
import copy

def successor_states(state, grid):
    def is_clear(location):
        return grid[location[0]][location[1]] != 1 and (location not
            in state['at-stone'] and location != state['at-player'])

    def move(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        if is_clear(new_player_location):
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            return new_state

    def push(direction):
        new_player_location = (state['at-player'][0] + direction[0],
            state['at-player'][1] + direction[1])
        new_stone_location = (new_player_location[0] + direction[0],
            new_player_location[1] + direction[1])
        if new_stone_location in state['at-stone'] and is_clear((
            new_stone_location[0] + direction[0], new_stone_location
            [1] + direction[1])) and (new_player_location[0] +
            direction[0], new_player_location[1] + direction[1]) not
            in state['at-stone']:
            new_state = copy.deepcopy(state)
            new_state['at-player'] = new_player_location
            stone_index = state['at-stone'].index(new_stone_location)
            new_state['at-stone'][stone_index] = (new_stone_location
                [0] + direction[0], new_stone_location[1] + direction
                [1])
            return new_state

    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
```

```
    successors = []
    for direction in directions:
        move_successor = move(direction)
        if move_successor:
            successors.append(move_successor)
        push_successor = push(direction)
        if push_successor and push_successor['at-player'] not in
            push_successor['at-stone']:
            successors.append(push_successor)
    return successors
```

```
The fix was in the push function, where I added an additional
    condition to check if the new player location is not already
    occupied by a stone. Also, I added a check before appending the
    push_successor to successors to ensure the player and stones are
    at different locations

System message:
Successor States Soundness Test Passed
Completeness test

System message:
Maximum iterations reached for completeness test. Process stopped.
```

9.5.5 Llama3.1 405B

Listing 64: Without partial successor soundness test

```
AutoToS prompt:

This domain models the classic Sokoban game, where the robot needs to
    move around and push stones to specific locations while adhering
    to the constraints defined by the walls in the maze.
The maze is defined by a grid of values 0,1, and 2, where 2 means it
    is a goal location for a stone, 1 means the cell is blocked, and
    either 0 or 2 means that the cell can be occupied. A cell is clear
     if it can be occupied, but is not occupied by either the player
    or any stone.

The actions are:

move: This action moves the player in a specified direction. To
    perform this action, the target location must be clear and
    adjacent to player's location. Performing this action will result
    in the player being at the target location and no longer at the
    starting location, meaning the starting location is now clear,
    while the target location is now not clear.

push: This action allows the player to push a stone in a specified
    direction. To perform this action, the player's location, the
    stone location, and the target location must line up in the same
    direction, and the target location must be clear. Performing the
    action results in player being at the stone's initial location and
     the stone is at the target location. Further, the player's
    previous location is clear, while the target location is not clear
    .

Assume that states are defined by a dictionary with keys at-player and
     at-stone.
Here is an example of a state
{'at-player': (4, 4), 'at-stone': [(2, 2), (3, 3)]}
Here is an example of the grid:
[[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1,
    1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0,
```

1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]

Provide a Python implementation of successor states function, receiving a state and a grid and returning a list of immediate successor states.

Model response:
Successor States Function
Python

```
def successor_states(state, grid):
    """
    This function generates all possible successor states for a given
        state in the Sokoban game.

    Args:
    state (dict): A dictionary containing the current state of the
        game.
                  It has two keys: 'at-player' and 'at-stone'.
                  'at-player' is a tuple representing the player's
                      location.
                  'at-stone' is a list of tuples, each representing a
                      stone's location.
    grid (list of lists): A 2D list representing the game grid.
                          It contains 0s (clear cells), 1s (blocked
                              cells), and 2s (goal locations).

    Returns:
    list of dicts: A list of dictionaries, each representing a
        possible successor state.
    """

    # Define possible movements (up, down, left, right)
    movements = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    # Function to check if a cell is within the grid boundaries
    def is_valid(cell):
        return 0 <= cell[0] < len(grid) and 0 <= cell[1] < len(grid
            [0])

    # Function to check if a cell is clear (not blocked and not
        occupied by the player or a stone)
    def is_clear(cell):
        return is_valid(cell) and grid[cell[0]][cell[1]] != 1 and cell
             not in state['at-stone'] and cell != state['at-player']

    # Initialize the list of successor states
    successors = []

    # Check all possible movements from the current player location
    for movement in movements:
        new_player_location = (state['at-player'][0] + movement[0],
            state['at-player'][1] + movement[1])

        # If the new player location is clear, add the new state to
            the successors list
        if is_clear(new_player_location):
            new_state = {'at-player': new_player_location, 'at-stone':
                 state['at-stone'].copy()}
            successors.append(new_state)

        # Check if the player can push a stone in the current
            direction
```

```
        stone_location = (new_player_location[0] + movement[0],
            new_player_location[1] + movement[1])
        if stone_location in state['at-stone']:
            new_stone_location = (stone_location[0] + movement[0],
                stone_location[1] + movement[1])
            if is_clear(new_stone_location):
                new_state = {'at-player': new_player_location, 'at-
                    stone': state['at-stone'].copy()}
                new_state['at-stone'].remove(stone_location)
                new_state['at-stone'].append(new_stone_location)
                successors.append(new_state)

    return successors
```

This function generates all possible successor states for a given state in the Sokoban game. It checks all possible movements from the current player location and adds the new states to the successors list if the new player location is clear. It also checks if the player can push a stone in the current direction and adds the new state to the successors list if the new stone location is clear.

Generated by Meta Llama 3.1-405

AutoToS prompt:

Provide only the python code that tests whether a state is a goal state for a given goal configuration. Assume that the goal configuration is provided by the 2-dimensional array as before. The function should receive a state and the 2-dimensional array and return True if the state is a goal state and False otherwise.

Model response:
Goal Test Function
Python

```
def is_goal_state(state, grid):
    """
    This function checks if a given state is a goal state.

    Args:
    state (dict): A dictionary containing the current state of the
        game.
                  It has two keys: 'at-player' and 'at-stone'.
                  'at-player' is a tuple representing the player's
                      location.
                  'at-stone' is a list of tuples, each representing a
                      stone's location.
    grid (list of lists): A 2D list representing the game grid.
                          It contains 0s (clear cells), 1s (blocked
                              cells), and 2s (goal locations).

    Returns:
    bool: True if the state is a goal state, False otherwise.
    """
    # Check if all stones are at goal locations
    for stone in state['at-stone']:
        if grid[stone[0]][stone[1]] != 2:
            return False

    return True
```

Generated by Meta Llama 3.1-405

System message:
Goal Soundness Test Passed

Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Iteration 1

AutoToS prompt:
Error occurred for grid configuration: [[1, 1, 1, 1, 1, 1], [1, 0, 0, 2, 0, 1], [1, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1], [1, 0, 1, 2, 1, 1], [1, 0, 0, 0, 1, 0], [1, 1, 1, 1, 1, 0]]
Successor function when run on the state {'at-player': (5, 3), 'at-stone': [(5, 2), (4, 3)]} failed to produce all successors.
Missing successors are: [{'at-player': (5, 2), 'at-stone': [(5, 1), (4, 3)]}]
First think step by step why the successor function failed to produce all successors of the state.
Then, fix the successor function.
Remember how you fixed the previous mistakes, if any. Keep the same function signature.

Model response:
Successor States Function (Updated)
Python
```
def successor_states(state, grid):
    """
    This function generates all possible successor states for a given
        state in the Sokoban game.

    Args:
    state (dict): A dictionary containing the current state of the
        game.
                  It has two keys: 'at-player' and 'at-stone'.
                  'at-player' is a tuple representing the player's
                      location.
                  'at-stone' is a list of tuples, each representing a
                      stone's location.
    grid (list of lists): A 2D list representing the game grid.
                          It contains 0s (clear cells), 1s (blocked
                              cells), and 2s (goal locations).

    Returns:
    list of dicts: A list of dictionaries, each representing a
        possible successor state.
    """

    # Define possible movements (up, down, left, right)
    movements = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    # Function to check if a cell is within the grid boundaries
    def is_valid(cell):
        return 0 <= cell[0] < len(grid) and 0 <= cell[1] < len(grid
            [0])

    # Function to check if a cell is clear (not blocked and not
        occupied by the player or a stone)
    def is_clear(cell):
        return is_valid(cell) and grid[cell[0]][cell[1]] != 1 and cell
             not in state['at-stone'] and cell != state['at-player']

    # Initialize the list of successor states
    successors = []

    # Check all possible movements from the current player location
```

```
    for movement in movements:
        new_player_location = (state['at-player'][0] + movement[0],
            state['at-player'][1] + movement[1])

        # If the new player location is clear, add the new state to
            the successors list
        if is_clear(new_player_location):
            new_state = {'at-player': new_player_location, 'at-stone':
                 state['at-stone'].copy()}
            successors.append(new_state)

        # Check if the player can push a stone in the current
            direction
        for stone in state['at-stone']:
            if (stone[0] == new_player_location[0] and stone[1] ==
                new_player_location[1]):
                new_stone_location = (stone[0] + movement[0], stone[1]
                     + movement[1])
                if is_clear(new_stone_location):
                    new_state = {'at-player': new_player_location, 'at
                        -stone': state['at-stone'].copy()}
                    new_state['at-stone'].remove(stone)
                    new_state['at-stone'].append(new_stone_location)
                    successors.append(new_state)

    return successors
```
```
The issue was that the previous function was only checking if the
    player could push a stone if the stone was in the stone_location
    which was calculated based on the new_player_location. However,
    this was not considering the case where the stone is already at
    the new_player_location. To fix this, we need to check for each
    stone in the state['at-stone'] if it is at the new_player_location
     and if the player can push it to the new_stone_location.
Now the function correctly generates all possible successor states for
     a given state in the Sokoban game.
Generated by Meta Llama 3.1-405

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Total success: 11, failure: 0, total expansions 763750, total
    generated 1955669
```

Listing 65: With partial successor soundness test

```
AutoToS prompt:

This domain models the classic Sokoban game, where the robot needs to
    move around and push stones to specific locations while adhering
    to the constraints defined by the walls in the maze.
The maze is defined by a grid of values 0,1, and 2, where 2 means it
    is a goal location for a stone, 1 means the cell is blocked, and
    either 0 or 2 means that the cell can be occupied. A cell is clear
     if it can be occupied, but is not occupied by either the player
    or any stone.

The actions are:

move: This action moves the player in a specified direction. To
    perform this action, the target location must be clear and
    adjacent to player's location. Performing this action will result
    in the player being at the target location and no longer at the
```

```
    starting location, meaning the starting location is now clear,
    while the target location is now not clear.

push: This action allows the player to push a stone in a specified
    direction. To perform this action, the player's location, the
    stone location, and the target location must line up in the same
    direction, and the target location must be clear. Performing the
    action results in player being at the stone's initial location and
     the stone is at the target location. Further, the player's
    previous location is clear, while the target location is not clear
    .

Assume that states are defined by a dictionary with keys at-player and
     at-stone.
Here is an example of a state
{'at-player': (4, 4), 'at-stone': [(2, 2), (3, 3)]}
Here is an example of the grid:
[[1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [1, 0, 0, 0, 1,
    1, 1, 1], [1, 0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 0, 0, 0, 0, 1], [0,
    1, 1, 1, 2, 2, 0, 1], [0, 0, 0, 1, 0, 1, 0, 1], [0, 0, 0, 1, 0, 0,
     0, 1], [0, 0, 0, 1, 1, 1, 1, 1]]

Provide a Python implementation of successor states function,
    receiving a state and a grid and returning a list of immediate
    successor states.

Model response:
Python
```

```
def successor_states(state, grid):
    """
    This function generates all possible successor states given a
        state and a grid.

    Parameters:
    state (dict): A dictionary containing the current state of the
        game. It has two keys: 'at-player' and 'at-stone'.
                  'at-player' is a tuple representing the current
                      position of the player.
                  'at-stone' is a list of tuples representing the
                      current positions of the stones.
    grid (list): A 2D list representing the game grid. It contains 0s,
         1s, and 2s where 0 represents an empty cell,
                 1 represents a blocked cell, and 2 represents a goal
                     location.

    Returns:
    list: A list of dictionaries representing all possible successor
        states.
    """

    # Define all possible movements (up, down, left, right)
    movements = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    # Function to check if a cell is clear
    def is_clear(cell):
        return grid[cell[0]][cell[1]] != 1 and cell not in state['at-
            stone'] and cell != state['at-player']

    # Function to check if a cell is a valid position for a stone
    def is_valid_stone_position(cell):
        return grid[cell[0]][cell[1]] in [0, 2]

    # Initialize the list of successor states
    successor_states = []
```

````
    # Check all possible movements for the player
    for movement in movements:
        new_player_position = (state['at-player'][0] + movement[0],
            state['at-player'][1] + movement[1])

        # Check if the new position is clear and within the grid
            boundaries
        if (0 <= new_player_position[0] < len(grid)) and (0 <=
            new_player_position[1] < len(grid[0])) and is_clear(
            new_player_position):
            # Create a new state with the updated player position
            new_state = {'at-player': new_player_position, 'at-stone':
                 state['at-stone'].copy()}

            # Add the new state to the list of successor states
            successor_states.append(new_state)

        # Check if the player can push a stone
        for stone in state['at-stone']:
            # Calculate the position of the stone after pushing
            new_stone_position = (stone[0] + movement[0], stone[1] +
                movement[1])

            # Check if the stone is adjacent to the player and the new
                 position is clear and within the grid boundaries
            if (stone[0] == state['at-player'][0] + movement[0] and
                stone[1] == state['at-player'][1] + movement[1] and
                (0 <= new_stone_position[0] < len(grid)) and (0 <=
                    new_stone_position[1] < len(grid[0])) and
                is_clear(new_stone_position) and
                    is_valid_stone_position(new_stone_position)):
                # Create a new state with the updated player and stone
                     positions
                new_state = {'at-player': stone, 'at-stone': state['at
                    -stone'].copy()}
                new_state['at-stone'].remove(stone)
                new_state['at-stone'].append(new_stone_position)

                # Add the new state to the list of successor states
                successor_states.append(new_state)

    return successor_states
```
Generated by Meta Llama 3.1-405

AutoToS prompt:

Provide only the python code that tests whether a state is a goal
    state for a given goal configuration. Assume that the goal
    configuration is provided by the 2-dimensional array as before.
    The function should receive a state and the 2-dimensional array
    and return True if the state is a goal state and False otherwise.

Model response:
Python
```
def is_goal_state(state, grid):
    """
    This function checks if a given state is a goal state.

    Parameters:
    state (dict): A dictionary containing the current state of the
        game. It has two keys: 'at-player' and 'at-stone'.
````

```
                  'at-player' is a tuple representing the current
                      position of the player.
                  'at-stone' is a list of tuples representing the
                      current positions of the stones.
    grid (list): A 2D list representing the game grid. It contains 0s,
         1s, and 2s where 0 represents an empty cell,
                 1 represents a blocked cell, and 2 represents a goal
                     location.

    Returns:
    bool: True if the state is a goal state, False otherwise.
    """
    # Check if all stones are at goal locations
    for stone in state['at-stone']:
        if grid[stone[0]][stone[1]] != 2:
            return False

    # If all stones are at goal locations, the state is a goal state
    return True
```

Generated by Meta Llama 3.1-405

System message:
Goal Soundness Test Passed
Soundness test

System message:
Successor States Soundness Test Passed
Completeness test
Successor Completeness Test Passed
Total success: 11, failure: 0, total expansions 763750, total generated 1955669