Team NUS-LIDG Technical Report: Embodied Agent Interface Challenge @ NeurIPS 2025

T. Duy Nguyen-Hien[1]* Wee Sun Lee[1]

[1]School of Computing
National University of Singapore
{duynht, leews}@comp.nus.edu.sg

Abstract

We study how far carefully designed language interfaces can take large language models (LLMs) on the Embodied Agent Interface (EAI) benchmark, which decomposes embodied decision making into goal interpretation, subgoal decomposition, action sequencing, and transition modeling in the BEHAVIOR and VirtualHome simulators. Rather than training new models, we keep LLMs fixed and redesign the prompts: we make predicate vocabularies and argument conventions explicit, enforce JSON schemas, and provide few-shot examples; for action sequencing, we additionally introduce a PDDL-based scaffold in which the LLM corrects approximate domain and problem files before either emitting a plan directly or delegating to FastDownward. Our results highlight interface design as a critical degree of freedom for embodied LLMs, even under fixed model weights.

1 Introduction

Large language models (LLMs) are increasingly used as *interfaces* between natural language and symbolic structures for embodied agents. The Embodied Agent Interface (EAI) benchmark [7] makes this explicit by decomposing embodied decision making into four modules – goal interpretation, subgoal decomposition, action sequencing, and transition modeling – evaluated in the BEHAVIOR [14, 6] and VirtualHome [9] simulators.

Most recent work varies the *models* (architectures, training, or data) while keeping the interface relatively ad hoc. In contrast, we adopt an explicitly *interface-centric* perspective: we keep the underlying LLMs fixed and modify only the prompts and symbolic scaffolding that sit between the benchmark data and the models. For goal interpretation, subgoal decomposition, and transition modeling, we treat the organizers' prompts as baselines and introduce refined templates that (i) make predicate vocabularies and argument conventions explicit, (ii) enforce strict JSON output schemas, and (iii) include carefully chosen few-shot examples. For action sequencing, we add a PDDL layer: instances are translated into approximate domain and problem files, which the LLM is asked to analyze and repair before either generating a PDDL plan or handing the corrected files to FastDownward [3].

This yields a simple question: *how far can we go in EAI by changing only the interface?* Our results show that interface design matters substantially. On BEHAVIOR, prompt refinement alone improves goal-interpretation F1 from 0.81 to 0.85, subgoal task success from 0.56 to 0.75, and transition-modeling F1 from 0.36 to 0.99; PDDL scaffolding further raises action-sequencing task success from 0.64 to 0.84. In VirtualHome, prompt refinement consistently improves goal interpretation, subgoals, and transition modeling, while PDDL-based action sequencing underperforms a simpler non-PDDL template.

*Correspondence: duynht@comp.nus.edu.sg

2 Related Work

Embodied benchmarks. Embodied AI has been studied in simulators such as AI2-THOR [5], iGibson [10], BEHAVIOR/BEHAVIOR-1K [14, 6], and VirtualHome [9], with tasks including navigation, manipulation, and long-horizon household activities. VirtualHome in particular represents activities as executable programs over symbolic actions [9]. The EAI benchmark [7] refines this line by factoring embodied decision making into four language-to-symbolic interfaces (goal interpretation, subgoals, action sequencing, transition modeling) and evaluating each by execution in BEHAVIOR and VirtualHome.

LLMs for robotics and planning. LLMs have been used as high-level planners or program synthesizers for robots, e.g., grounding language in affordance models [1], generating executable policy code [8], and producing program-like plans [13, 4]. A complementary line couples LLMs with PDDL and classical planners: using LLMs as few-shot planners or heuristics [12], constructing or debugging PDDL world models [2, 11], or probing their planning limitations [15], including multi-agent settings [16].

Our focus. We do not introduce new models or training procedures. Instead, we treat the *interface* itself as the object of study: a unified set of prompt templates for the four EAI tasks across two environments, plus a lightweight PDDL scaffold for action sequencing. We then systematically compare (i) prompt-only refinements of the official templates and (ii) PDDL-based scaffolding, highlighting when each helps or hurts in embodied evaluation.

3 Method

We treat each EAI subtask as a black-box *language interface* between the benchmark data and an LLM. The underlying models (GPT-5, GPT-5.1, and Gemini models) are fixed; we vary only (i) the prompt templates and (ii) an optional PDDL scaffold for action sequencing. For every instance, our code reads the organizers' JSON input, instantiates prompts from our templates, queries an LLM, and converts the response back into the official EAI format using the provided evaluation scripts [7].

The output is required to be valid JSON and to satisfy the schemas expected by the EAI evaluation code. Across templates, we differ only in: (a) how much of the predicate vocabulary and argument conventions is made explicit to the model, (b) how strongly the output format is constrained, and (c) whether we introduce auxiliary structures such as PDDL domains and problems.

3.1 Goal Interpretation

The goal interpretation task maps natural language instructions to symbolic goal states. In BEHAVIOR, these goal states are represented as unary "node" goals over object states and binary "edge" goals over object–object relations. The organizers' original prompt template (Template 1) engages the LLM in a "helpful assistant" persona, describes the task informally, and asks the model to output lists of predicates. Although essential components such as the full predicate vocabulary and output format are present, they are not organized into clearly separated sections.

Instead of the "helpful assistant" persona, our BEHAVIOR template (Template 2) adopts the persona of "an expert in symbolic AI". We reframe the problem as a structured translation task and leverage markdown-style structure to organize dedicated sections for the symbolic representation format, input format, and output format. Our template explicitly enumerates the available node predicates (e.g., `Cooked`, `Burnt`, `Dusty`, `Frozen`, `Open`, `Sliced`, `Soaked`, `Stained`, `ToggledOn`, `Holds_Rh`, `Holds_Lh`) and edge predicates (e.g., `OnTop`, `NextTo`, `Inside`, `OnFloor`, `Touching`, `Under`), and presents their argument signatures and informal descriptions in a table, inspired by the organizers' Template 7 for the subgoal decomposition task. We use a two-shot prompting strategy with worked examples, showing how to translate instructions such as "clean the barbecue grill" and "clean the freezer" into combinations of predicates as symbolic goals. We enforce a strict JSON output schema, require that all object identifiers come from the provided relevant-object list, and specify that negation must be written as nested lists (e.g., `["not",`

`["Stained", "cup.n.01_1"]`) to align with the EAI evaluation tooling [7].

In VirtualHome, the interface is extended with *action goals* that capture whether specific actions must occur, in addition to satisfying state predicates [9, 7]. Templates 4 and 5 adopt a similar prompting strategy to BEHAVIOR, with additional instructions for handling action goals. In particular, action goals are instructed to correspond to the "unambiguously persistent ongoing action of the character at the final goal state". Because the domain description is long – covering symbolic representation, scene details, input format, and output format – we opt for a one-shot strategy to save context length, and compensate by including a worked example with a detailed explanation.

We also observed inconsistencies between natural-language instructions and available objects in a scene, such as the use of "fridge" versus "freezer", and goal annotations that conflict with common sense (e.g., clothes required to be *on* rather than *inside* a washing machine). These inconsistencies sometimes lead to poor performance due to the LLM producing common-sense states. Template 5 yields improved performance over Template 4 by explicitly encoding inconsistency patching rules (Snippet 1).

Template 6 adds one additional special rule (Snippet 2) to prune the output symbolic goals to the subset directly relevant to the task name and final instruction. This improves development set performance. However, the results are not submitted for evaluation on the held-out test set since they were only available after the competition ended.

3.2 Subgoal Decomposition

Subgoal decomposition takes the initial state and final goals as input and outputs a temporally ordered list of intermediate symbolic states that should be achieved on the way to the goal. We design prompts that treat each subgoal as the effect of an implicit action, without forcing the model to name the action explicitly; this approach is compatible with both BEHAVIOR and VirtualHome [7].

Our BEHAVIOR subgoal prompts start from the organizers' Template 7, which already introduces a rich vocabulary of state predicates and logical connectives but does not specify a named persona. Template 8 switches to a persona of "an expert in symbolic AI working with a robot", clarifies how state changes such as cleaning, soaking, and slicing can be achieved, and explicitly encodes hand-occupancy constraints via `holds_rh` and `holds_lh`. We require the LLM to reason about possible "pseudo-actions" needed for an action plan that would traverse the subgoal path, and to perform implicit planning to sanity-check the proposed subgoals.

In Template 9 and Template 10, we continue enriching the state descriptions (e.g., multi-step recipes for soaking and stain removal, guidance on when containers need to be opened) while progressively restricting the allowed connectives in the *output* to `and`/`or`/`not`; other quantifiers are allowed for internal reasoning but must not appear in the final JSON. Finally, in Template 11 we update the persona to an "expert in symbolic AI and PDDL" and add more explicit precondition–effect patterns for cleaning, soaking, freezing, opening containers, and manipulating containment (e.g., removing stains via `toggled_on(sink)` followed by `inside(obj, sink)`, or using soaked cleaning tools), while emphasizing that any `forall`/`exists`/counting expressions must be expanded into concrete object-specific predicates in the output.

For VirtualHome, Template 14 adapts the same approach. The organizers' template (Template 12) already provides a fairly complete specification with competitive performance, enumerating state and action primitives and explaining their argument and type constraints [7]. We therefore modify it only mildly in Template 13, adding a sentence in the persona section describing the arity of predicates (unary state predicates, binary relation predicates, and action predicates) and emphasizing the limited use of action goals: *"Please do not use actions in your output if you can fully describe goal states with allowable state primitives only."* The action section reinforces this restriction: *"Only use actions when it is impossible to fully describe goal states with allowable state primitives alone."*

3.3 Action Sequencing

For action sequencing, we introduce an intermediate PDDL representation rather than directly prompting for the final output format. For each domain, a PDDL domain file is manually drafted

from the organizers' tutorial repository[1] and is therefore approximate and incomplete. Each task instance is then parsed and transformed into a PDDL problem that encodes the initial and goal states based on the tutorial materials.

Template 16 provides the LLM with a candidate domain and problem, along with worked examples of domain corrections (e.g., adding identity predicates to prevent grasping floors, fixing typing constraints, and refining preconditions for cleaning actions). The LLM is instructed to act as a PDDL analyzer and debugger and to output a JSON object containing fields `"corrected_domain"`, `"corrected_problem"`, and `"action_plan"`, where `"action_plan"` is a list of PDDL actions such as `"(navigate_to sink_n_01_1 agent_n_01_1)"`. This design is inspired by work that uses LLMs to construct or repair PDDL models before handing them to classical planners [12, 2, 11].

We evaluate two planning approaches: (i) *LLM-PDDL*, which directly lets the LLM generate a PDDL action plan and parses it into the EAI output format; and (ii) *LLM+FastDownward*, which uses the corrected domain and problem as input to FastDownward with A* search to obtain a symbolic plan, which is then parsed into the same format. This mirrors prior studies that use LLMs both as stand-alone planners and as guides or model constructors for classical planning [12, 2, 15].

In VirtualHome, Template 19 follows the same design. To facilitate action goals, we instruct the LLM to modify the PDDL domain to incorporate auxiliary goal predicates denoting the execution of target actions, such as `did_watch` for the action `watch`, whenever action goals are expected.

For comparison, we also use non-PDDL templates from the organizers (Template 17) and a slight modification (Template 18) that prompts the LLM to output action strings directly in the EAI format. These direct templates resemble earlier LLM-for-robotics systems that generate action sequences without an explicit PDDL scaffold [1, 8, 13].

3.4 Transition Modeling

Transition modeling requires predicting the preconditions and effects of symbolic actions necessary to reach the goal states from initial states [7]. The EAI benchmark decomposes this into multiple metrics over object affordances, spatial relations, non-spatial relations, object orientations, and object states.

In this task, our prompt templates instruct the LLM with the algorithmic strategy in Snippet 3. We include in each template a one-shot example with step-by-step explanation following this algorithm. The template variants experiment with exposing the gold action set from the tutorial repository. For BEHAVIOR, Template 21 includes no gold actions, while Template 22 follows the organizers' approach in Template 20 by inserting a "commonly used actions" snippet that contains all BEHAVIOR actions from the tutorial repository.

Similarly, Template 24 introduces the algorithmic instructions with a one-shot example for VirtualHome but no common actions; Template 25 adds all VirtualHome gold actions; and Template 26 further emphasizes that these common actions should be reused *in full* with Snippet 4.

3.5 Environment-Specific Adaptations

Our prompts share a common structure across BEHAVIOR and VirtualHome but employ environment-specific predicate vocabularies, examples, and PDDL domains. In BEHAVIOR, we focus on object-centric states and simple agent reachability predicates, and rely heavily on cleaning and manipulation affordances [14, 6]. In VirtualHome, we adapt the templates to character-centric predicates (e.g., `Sitting`, `Lying`, `Facing`), room membership (`inside_room`), and device-related states (`plugged_in`, `on`, `off`) If additional environment-specific heuristics or preprocessing (e.g., object type remapping, automatic scene filtering) are used, they can be encoded as a thin symbolic layer that feeds into the same prompt templates; in our implementation we only rely on the information provided by the official EAI data.

[1] https://github.com/embodied-agent-interface/embodied-agent-interface

4 Implementation Details

We implement each task as a separate prompting pipeline that reads the organizers' JSON-formatted instances, constructs the appropriate prompt template, queries an LLM, and parses the JSON output back into the EAI interface using the official evaluation scripts [7]. All templates used in the reported results are included in the appendix.

Our primary results are derived from GPT-5 (`gpt-5-2025-08-07`) with the reasoning effort set to `medium`, unless specified otherwise, and a maximum output length of 128,000 tokens. We do not control sampling parameters (e.g., temperature, top-p) as the current API does not support their customization for GPT-5.

We also conduct supplementary experiments with GPT-5.1 (`gpt-5.1-2025-11-13`), Gemini-Robotics-ER-1.5 (`gemini-robotics-er-1.5-preview`), and Gemini-3-Pro (`gemini-3-pro-preview`). These models are accessed via OpenAI-compatible APIs using a similar `medium` reasoning effort. For the Gemini models, this setting maps to `thinking_budget=8192` (Gemini-Robotics-ER-1.5) and `thinking_level='high'` (Gemini-3-Pro) and uses the default temperature of 1.0 within Gemini's $[0, 2]$ range, the default top-p of 0.95, and a maximum output length of 65,536 tokens. Due to API rate limits and resource constraints, these results are less comprehensive, but we include them for completeness.

Post-processing consists of robust JSON parsing and conversion to the exact formats expected by the EAI evaluation scripts, with simple fallbacks for minor formatting issues.

For action sequencing with FastDownward, we use a Python wrapper to feed the LLM-corrected PDDL domain and problem strings to FastDownward, which employs A* search with unit costs. The resulting symbolic plan is converted to the EAI output format, while filtering out actions that are admissible in PDDL but not allowed by the EAI simulator interface (e.g., navigations).

5 Evaluation Results

The competition organizers provided us with a BEHAVIOR development set and a VirtualHome development set, together with an online leaderboard. Only the development sets can be evaluated locally using the code in the official tutorial repository. Leaderboard evaluation uses the same BEHAVIOR development set as input, whereas for VirtualHome the server evaluates on a subset of the development set combined with a held-out test set. Accordingly, we report two types of metrics: *Dev*, computed locally on the development split, and *Dev+Test*, which denotes leaderboard scores on the hidden server evaluation (VirtualHome only).

For VirtualHome, our Dev numbers are computed on the overlap between the public development set and the leaderboard split, namely examples whose identifiers begin with `scene_1` in the evaluation phase prompts, rather than on the full development set. For BEHAVIOR, all tables report local development set results. During the challenge, the organizers updated the evaluation code used on the leaderboard that has not yet been reflected in the public tutorial repository, which causes our local BEHAVIOR subgoal decomposition task success rate (Task SR) to be about 0.02 lower than the corresponding leaderboard values. For consistency with ablations and error breakdowns that are only available with the local evaluator, we nevertheless report the local development numbers. We present in this sections summary statistics, while the full result breakdowns are available in Appendix B.

5.1 BEHAVIOR

Table 1 reports BEHAVIOR goal interpretation performance.

Table 2 summarizes BEHAVIOR subgoal decomposition. We report task success (Task SR) and trajectory execution success (Exec. SR) for representative prompt variants.

Table 3 reports BEHAVIOR action sequencing results, highlighting the effect of the PDDL scaffold.

Finally, Table 4 shows BEHAVIOR transition modeling performance, aggregated over object

states, spatial relations, and non-spatial relations.

Table 1: BEHAVIOR goal interpretation.

Method	Overall F1
GPT-5 + Template 1 (Baseline)	0.807
GPT-5 + Template 2	**0.847**
Gemini-Robotics-ER-1.5 + Template 2	0.841

Table 2: BEHAVIOR subgoal decomposition.

Method	Task SR	Exec. SR
GPT-5 + Template 7 (Baseline)	0.560	0.640
GPT-5 + Template 9	0.700	0.830
GPT-5 + Template 10	0.730	**0.860**
GPT-5 + Template 11	0.740	**0.860**
Gemini-3-Pro + Template 11	**0.750**	0.840

Table 3: BEHAVIOR action sequencing.

Method	Task SR	Exec. SR
GPT-5 + Template 15 (Baseline)	0.640	0.680
GPT-5 + Template 16 (PDDL)	**0.840**	**0.900**
GPT-5 + Template 16 (PDDL) + FastDownward	0.740	0.820

Table 4: BEHAVIOR transition modeling.

Method	Overall F1	Overall SR
GPT-5 + Template 20 (Baseline)	0.364	**0.980**
GPT-5 + Template 21	0.515	0.970
GPT-5 + Template 22	**0.993**	**0.980**

5.2 VirtualHome

For VirtualHome goal interpretation, Table 5 reports both development and Dev+Test aggregates.

Table 6 reports VirtualHome subgoal decomposition performance.

VirtualHome action sequencing results are summarized in Table 7, comparing direct prompting and PDDL-based variants.

Finally, Table 8 reports VirtualHome transition modeling performance on both the development split and the Dev+Test leaderboard split.

Overall, these results show that prompt refinement reliably improves performance across tasks and environments, and that PDDL scaffolding yields large gains for BEHAVIOR action sequencing, while being more brittle in VirtualHome where the approximate domain is less faithful to the simulator.

6 Analysis and Discussion

Goal interpretation. Although both environments share the same general approach and model (GPT-5), results clearly show that the LLM struggles more in the VirtualHome domain. The best Dev+Test overall F1 is 0.464, compared to BEHAVIOR where even the baseline achieves 0.807. On

Table 5: VirtualHome goal interpretation.

Method	Dev F1	Dev+Test F1
GPT-5 + Template 3 (Baseline)	0.466	0.366
GPT-5 + Template 4	0.606	0.432
GPT-5 + Template 5	0.645	**0.464**
GPT-5 + Template 6	**0.663**	—

Table 6: VirtualHome subgoal decomposition.

	Dev		Dev+Test	
Method	Task SR	Exec. SR	Task SR	Exec. SR
GPT-5 + Template 12 (Baseline)	0.898	0.927	0.742	0.886
GPT-5 + Template 13	**0.906**	0.935	**0.747**	0.887
GPT-5 + Template 14	0.902	**0.942**	**0.747**	**0.901**

Table 7: VirtualHome action sequencing.

	Dev		Dev+Test	
Method	Task SR	Exec. SR	Task SR	Exec. SR
GPT-5 + Template 17 (Baseline)	**0.790**	0.852	0.685	0.826
GPT-5 + Template 18	**0.790**	0.849	0.727	0.828
GPT-5 + Template 19 (PDDL)	0.433	0.472	0.447	0.481
GPT-5 + Template 19 (PDDL) + FD	0.410	0.462	0.424	0.473
GPT-5 + Template 19 (PDDL) + `PLUGIN` deleted	0.754	**0.856**	**0.747**	**0.834**
GPT-5 + Template 19 (PDDL) + FD + `PLUGIN` deleted	0.682	0.827	0.668	0.758

Table 8: VirtualHome transition modeling.

	Dev		Dev+Test	
Method	Overall F1	Overall SR	Overall F1	Overall SR
GPT-5 + Template 23 (Baseline)	0.349	0.909	0.344	0.868
GPT-5 + Template 24	0.459	0.946	0.440	0.952
GPT-5 + Template 25	0.821	0.983	0.825	0.994
GPT-5 + Template 26	**0.994**	**0.997**	**0.996**	**0.998**

the VirtualHome development set, the model achieves recall above 0.80, but precision is substantially lower, indicating that the model tends to output more predicates than appear in the ground truth. For example, in instance `scene_1_429_1`, the goal name "Pet cat" and description "I see my cat on the couch so I walk over, sit down and pet the cat" only require the relation that the character is `CLOSE` to the cat and the action `TOUCH` as ground truth. When the LLM additionally outputs a state goal such as "character `ON` couch", this becomes a false positive under the evaluation metric. We also observe examples where it would be reasonable to include non-empty state and relation goals but the ground truth only admits empty sets, which hurts both precision and recall and highlights some annotation mismatches.

Subgoal decomposition. In contrast to goal interpretation, LLMs perform relatively well on subgoal decomposition in both environments. In BEHAVIOR, LLMs still suffer from *additional steps* runtime errors in which they repeat previously satisfied subgoals (e.g., soaking a rag multiple times or

cleaning an already clean grill), but overall task and trajectory execution success rates increase noticeably with our refined templates. In VirtualHome, the rate of additional-step errors on the development set for GPT-5 exceeds 0.25, but the overall task and trajectory execution success remain high (Table 6), suggesting that the extra steps often correspond to redundant but harmless actions rather than catastrophic failures.

Action sequencing. Our PDDL prompting strategy substantially improves performance in BEHAVIOR but underperforms direct prompting in VirtualHome. In BEHAVIOR, the PDDL plans produced by the LLM even surpass those obtained by the FastDownward planner when both use the same corrected domain and problem, echoing observations in prior PDDL+LLM work that LLMs can be surprisingly robust to incomplete or noisy domain specifications when combined with commonsense priors [12, 2, 11]. However, there appears to be a threshold of tolerable domain mismatch: in VirtualHome, the seed PDDL domain is larger and more complex, and includes action goals. We observe a recurring pathology where the LLM negates initial `plugged_in` states into "dormant" defaults (e.g., making objects unplugged), despite explicit instructions not to do so, leading to substantial hallucination errors. By indiscriminately deleting all `PLUGIN` actions in the output, the PDDL-based methods experience a large bump in performance (Table 7), but this is an ad-hoc fix. These findings resonate with prior reports that LLMs used directly as planners are brittle, and that careful domain modeling and error-checking are crucial [15, 12, 2].

Transition modeling. Guiding GPT-5 with the inversion-style algorithm in the prompt already yields strong planning success rates in both domains. However, the F1-score for exactly matching ground-truth preconditions and effects does not improve as much until we expose the model to the full set of gold actions. Manual inspection of the development set shows that the LLM sometimes outputs *more complete* action definitions than the annotations. For instance, `walk_towards` might be defined so that the agent becomes near the target object and other objects that are near, inside, under, or on top of it, whereas the gold action only updates a subset of these predicates. Our experiments with the "commonly used actions" snippet reveal that the transition modeling metrics can be trivially satisfied if the goal actions are included in the prompt template. In VirtualHome, where GPT-5 tends to abbreviate long action definitions to save on the token budget after having already consumed a large prompt, unless explicitly instructed to reuse them in full (Snippet 4), at which point metrics become near-perfect. This call for a more robust evaluation design for this transition modeling, for example by strictly having held-out action definitions for the test set.

7 Conclusion

We presented a prompt-centric approach to the Embodied Agent Interface Challenge that treats LLMs as symbolic interface learners rather than end-to-end policy generators. By refining the organizers' prompts, explicitly documenting predicate vocabularies, enforcing JSON schemas, introducing PDDL-based scaffolding for planning, and algorithmic prompting for transition modeling, we obtained substantial gains over baseline interfaces in BEHAVIOR and VirtualHome. Our results show that careful interface design can unlock much of the potential of general-purpose LLMs for embodied reasoning without any fine-tuning.

At the same time, the divergence between BEHAVIOR and VirtualHome action sequencing highlights the limitations of approximate PDDL domains and the sensitivity of performance to mismatches between symbolic representations and simulator affordances. Future work could explore learning or refining domains from data [2, 11], integrating environment feedback into prompt selection [4], and combining classical planning with LLMs in ways that are robust to partial or noisy domain knowledge [12, 15]. More broadly, we believe that treating interface design as a first-class research problem – alongside perception, control, and model design – have high potential in building reliable embodied agents.

Reproducibility Statement

Our implementation builds directly on the official EAI codebase and evaluation scripts [7]. All prompt templates used in this report are included in the appendix. We use off-the-shelf API access to GPT-5, GPT-5.1 variants, Gemini-Robotics-ER-1.5, and Gemini-3-Pro with task-specific prompts but without fine-tuning. All implementation details are described in Section 4. We plan to release our code and configuration files after the competition to facilitate verification and reuse.

Acknowledgments

We thank the Embodied Agent Interface organizers for releasing the benchmark and codebase. This research is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG3-PhD 2023-08-053).

References

[1] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

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

[3] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[4] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022.

[5] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Matt Deitke, Kiana Ehsani, Daniel Gordon, Yuke Zhu, et al. Ai2-thor: An interactive 3d environment for visual ai. *arXiv preprint arXiv:1712.05474*, 2017.

[6] Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Michael Lingelbach, Jiankai Sun, et al. Behavior-1k: A benchmark for embodied ai with 1,000 everyday activities and realistic simulation. In *Conference on Robot Learning*, pages 80–93. PMLR, 2023.

[7] Manling Li, Shiyu Zhao, Qineng Wang, Kangrui Wang, Yu Zhou, Sanjana Srivastava, Cem Gokmen, Tony Lee, Erran Li Li, Ruohan Zhang, et al. Embodied agent interface: Benchmarking llms for embodied decision making. *Advances in Neural Information Processing Systems*, 37:100428–100534, 2024.

[8] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.

[9] Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. Virtualhome: Simulating household activities via programs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[10] Bokui Shen, Fei Xia, Chengshu Li, Roberto Martín-Martín, Linxi Fan, Guanzhi Wang, Claudia Pérez-D'Arpino, Shyamal Buch, Sanjana Srivastava, Lyne Tchapmi, et al. igibson 1.0: A simulation environment for interactive tasks in large realistic scenes. In *2021 IEEE/RSJ*

International Conference on Intelligent Robots and Systems (IROS), pages 7520–7527. IEEE, 2021.

[11] Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B Tenenbaum, Leslie Kaelbling, and Michael Katz. Generalized planning in pddl domains with pretrained large language models. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 20256–20264, 2024.

[12] Tom Silver, Varun Hariprasad, Reece S Shuttleworth, Nishanth Kumar, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Pddl planning with pretrained large language models. In *NeurIPS 2022 foundation models for decision making workshop*, 2022.

[13] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.

[14] Sanjana Srivastava, Chengshu Li, Michael Lingelbach, Roberto Martín-Martín, Fei Xia, Kent Elliott Vainio, Zheng Lian, Cem Gokmen, Shyamal Buch, Karen Liu, et al. Behavior: Benchmark for everyday household activities in virtual, interactive, and ecological environments. In *Conference on robot learning*, pages 477–490. PMLR, 2022.

[15] Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Large language models still can't plan (a benchmark for llms on planning and reasoning about change). In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.

[16] Xiaopan Zhang, Hao Qin, Fuquan Wang, Yue Dong, and Jiachen Li. Lamma-p: Generalizable multi-agent long-horizon task allocation and planning with lm-driven pddl planner. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10221–10221. IEEE, 2025.

A Biography of all team members

Team name: NUS-LIDG.

T. Duy Nguyen-Hien is a Ph.D. candidate at the School of Computing, National University of Singapore. His research interests include uncertainty quantification and calibration, decision-making under uncertainty, and probabilistic reasoning with large language models.

Wee Sun Lee is a Professor of Computer Science at the School of Computing, National University of Singapore. His research spans machine learning, planning, and probabilistic inference, with a focus on decision-making under uncertainty, Monte Carlo methods, and applications to robotics and AI planning.

B Full Results

B.1 BEHAVIOR

B.2 VirtualHome

Table 9: BEHAVIOR goal interpretation – full metrics.

	Relation Goal			State Goal			Overall		
Method	P	R	F1	P	R	F1	P	R	F1
GPT-5 + Template 1 (Baseline)	0.774	0.798	0.786	0.798	0.961	0.872	0.781	0.835	0.807
GPT-5 + Template 2	0.829	0.850	0.840	0.806	0.948	0.871	0.823	0.872	0.847
Gemini-Robotics-ER-1.5 + Template 2	0.886	0.788	0.834	0.835	0.895	0.864	0.872	0.813	0.841

P: Precision; R: Recall

Table 10: BEHAVIOR subgoal decomposition – full metrics.

	Success Rates					Grammar Errors			Runtime Errors			
Method	Task	St. Goals	Rel. Goals	All Goals	Traj. Exec.	Parse	Hallu.	# Arg.	Order	Miss.	Afford.	Add.
GPT-5 + Template 7 (Baseline)	0.56	0.62	0.569	0.583	0.64	0.00	0.01	0.00	0.04	0.30	0.01	0.05
GPT-5 + Template 9	0.70	0.75	0.832	0.810	0.83	0.00	0.01	0.00	0.06	0.06	0.04	0.17
Gemini-Robotics-ER-1.5 + Template 9	0.57	0.51	0.730	0.670	0.62	0.21	0.01	0.00	0.04	0.12	0.00	0.06
GPT-5.1 + Template 9	0.61	0.76	0.764	0.763	0.76	0.02	0.01	0.00	0.04	0.14	0.03	0.14
GPT-5 + Template 10	0.73	0.82	0.809	0.812	0.86	0.03	0.01	0.00	0.06	0.01	0.03	0.12
GPT-5-High + Template 10	0.73	0.88	0.830	0.842	0.85	0.02	0.01	0.00	0.06	0.04	0.02	0.12
Gemini-Robotics-ER-1.5 + Template 10	0.48	0.475	0.601	0.567	0.53	0.19	0.02	0.00	0.05	0.20	0.01	0.05
GPT-5 + Template 11	0.74	0.88	0.820	0.837	0.86	0.01	0.01	0.00	0.07	0.04	0.01	0.13
Gemini-3-Pro + Template 11	0.75	0.73	0.876	0.837	0.84	0.00	0.01	0.00	0.11	0.01	0.03	0.08

St.: State; Rel.: Relation; Traj. Exec.: Trajectory Execution; Hallu.: Hallucination; # Arg.: Number of Arguments; Miss.: Missing Steps; Afford.: Affordance; Add.: Additional Steps

Table 11: BEHAVIOR action sequencing – full metrics.

	Success Rates					Grammar Errors			Runtime Errors			
Method	Task	St. Goals	Rel. Goals	All Goals	Traj. Exec.	Parse	Hallu.	# Arg.	Order	Miss.	Afford.	Add.
GPT-5 + Template 15 (Baseline)	0.640	0.740	0.576	0.621	0.680	0.000	0.000	0.000	0.000	0.260	0.040	0.040
GPT-5 + Template 16	0.840	0.830	0.891	0.874	0.900	0.010	0.000	0.000	0.030	0.010	0.050	0.000
GPT-5 + Template 16 + FD	0.740	0.840	0.733	0.762	0.820	0.090	0.000	0.000	0.000	0.030	0.060	0.000

St.: State; Rel.: Relation; Traj. Exec.: Trajectory Execution; Hallu.: Hallucination; # Arg.: Number of Arguments; Miss.: Missing Steps; Afford.: Affordance; Add.: Additional Steps

Table 12: BEHAVIOR transition modeling – full metrics.

	Object States				Spatial Relations				Non-spatial Relations				Overall			
Method	P	R	F1	SR	P	R	F1	SR	P	R	F1	SR	P	R	F1	SR
GPT-5 + Template 20 (Baseline)	0.159	0.130	0.143	0.956	0.409	0.218	0.285	0.989	0.571	0.521	0.546	0.984	0.438	0.312	0.364	0.980
GPT-5 + Template 21	0.483	0.404	0.440	0.956	0.426	0.328	0.371	0.989	0.754	0.689	0.720	0.951	0.566	0.472	0.515	0.970
GPT-5 + Template 22	0.991	0.991	0.991	0.978	0.993	0.992	0.993	0.989	0.996	0.992	0.994	0.967	0.994	0.992	0.993	0.980

P: Precision; R: Recall; SR: Success Rate

Table 13: VirtualHome goal interpretation – full metrics.

	Dev												Dev+Test
	Relation Goal			State Goal			Action Goal			Overall			
Method	P	R	F1	P	R	F1	P	R	F1	P	R	F1	Overall F1
GPT-5 + Template 3 (Baseline)	0.460	0.658	0.541	0.341	0.771	0.473	0.246	0.846	0.381	0.340	0.744	0.466	0.366
GPT-5 + Template 4	0.516	0.691	0.591	0.470	0.821	0.598	0.701	0.636	0.667	0.516	0.735	0.606	0.432
GPT-5 + Template 5	0.569	0.859	0.684	0.485	0.800	0.604	0.698	0.642	0.669	0.545	0.790	0.645	0.464
GPT-5 + Template 6	0.698	0.792	0.742	0.571	0.559	0.565	0.766	0.667	0.713	0.658	0.668	0.663	—

P: Precision; R: Recall

Table 14: VirtualHome subgoal decomposition – full metrics.

	Dev										Dev+Test	
	Success Rates						Grammar Error	Runtime Error			Success Rates	
Method	Task	St. Goals	Rel. Goals	Act. Goals	All Goals	Traj. Exec.	Hallu.	Miss.	Afford.	Add.	Task	Traj. Exec.
GPT-5 + Template 12 (Baseline)	0.898	0.931	0.894	0.896	0.910	0.927	0.022	0.029	0.022	0.269	0.742	0.886
GPT-5 + Template 13	0.906	0.939	0.903	0.888	0.915	0.935	0.022	0.029	0.015	0.247	0.747	0.887
GPT-5 + Template 14	0.902	0.942	0.879	0.918	0.913	0.942	0.018	0.033	0.007	0.255	0.747	0.901

Grammar Error Parse, Grammar Error # Arg., and Runtime Error Order are all 0 and are omitted to save space.
St.: State; Rel.: Relation; Act.: Action; Traj. Exec.: Trajectory Execution; # Arg.: Number of Arguments; Miss.: Missing Steps; Afford.: Affordance; Add.: Additional Steps

Table 15: VirtualHome action sequencing – full metrics.

	Dev													Dev+Test	
	Success Rates						Grammar Errors			Runtime Errors				Success Rates	
Method	Task	St. Goals	Rel. Goals	Act. Goals	All Goals	Traj. Exec.	Parse	Hallu.	# Arg.	Order	Miss.	Afford.	Add.	Task	Traj. Exec
GPT-5 + Template 17 (Baseline)	0.790	0.932	0.811	0.689	0.837	0.852	0.003	0.013	0.000	0.036	0.918	0.000	0.013	0.685	0.826
GPT-5 + Template 18	0.790	0.917	0.800	0.709	0.832	0.849	0.007	0.013	0.007	0.036	0.089	0.000	0.013	0.727	0.828
GPT-5 + Template 19	0.433	0.148	0.678	0.500	0.391	0.472	0.000	0.426	0.007	0.003	0.082	0.010	0.000	0.447	0.481
GPT-5 + Template 19 + FD	0.410	0.165	0.650	0.453	0.380	0.462	0.656	0.397	0.023	0.010	0.095	0.007	0.013	0.424	0.473
GPT-5 + Template 19 + `PLUGIN` del	0.754	0.856	0.850	0.689	0.814	0.807	0.000	0.000	0.007	0.020	0.154	0.013	0.016	0.747	0.834
GPT-5 + Template 19 + FD + `PLUGIN` del	0.682	0.827	0.783	0.622	0.764	0.754	0.007	0.010	0.023	0.010	0.190	0.007	0.026	0.668	0.758

St.: State; Rel.: Relation; Act.: Action; Traj. Exec.: Trajectory Execution; Hallu.: Hallucination; # Arg.: Number of Arguments; Miss.: Missing Steps; Afford.: Affordance; Add.: Additional Steps

Table 16: VirtualHome transition modeling (Dev) – Part 1.

	Object Affordances				Spatial Relations				Non-spatial Relations			
Method	P	R	F1	SR	P	R	F1	SR	P	R	F1	SR
GPT-5 + Template 23 (Baseline)	0.311	0.331	0.320	0.910	0.310	0.311	0.311	0.897	0.137	0.078	0.099	0.905
GPT-5 + Template 24	0.388	0.363	0.375	0.928	0.576	0.393	0.468	0.972	0.304	0.170	0.218	0.973
GPT-5 + Template 25	0.888	0.892	0.890	0.982	0.895	0.694	0.782	0.986	0.727	0.584	0.648	0.986
GPT-5 + Template 26	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.993	0.998	0.997	0.998	1.000

P: Precision; R: Recall; SR: Success Rate

Table 17: VirtualHome transition modeling (Dev) – Part 2.

	Object Orientation				Object States				Overall			
Method	P	R	F1	SR	P	R	F1	SR	P	R	F1	SR
GPT-5 + Template 23 (Baseline)	0.146	0.625	0.235	0.893	0.673	0.385	0.490	0.921	0.392	0.314	0.349	0.909
GPT-5 + Template 24	0.000	0.000	0.000	1.000	0.579	0.527	0.552	0.921	0.519	0.411	0.459	0.946
GPT-5 + Template 25	0.970	1.000	0.985	1.000	0.892	0.870	0.881	0.978	0.872	0.776	0.821	0.983
GPT-5 + Template 26	1.000	1.000	1.000	1.000	0.987	0.987	0.987	0.994	0.994	0.994	0.994	0.997

P: Precision; R: Recall; SR: Success Rate

Table 18: VirtualHome transition modeling (Dev+Test) – overall.

Method	Overall F1	Overall SR
GPT-5 + Template 23 (Baseline)	0.344	0.868
GPT-5 + Template 24	0.440	0.952
GPT-5 + Template 25	0.825	0.994
GPT-5 + Template 26	0.996	0.998

SR: Success Rate

C Prompt Snippets

Snippet 1: VirtualHome Inconsistency Patching Rules

```
# SPECIAL RULES
* Note that the "fridge" and the "freezer" often means the same
    thing to the robot and should be use interchangeably when the
    natural language command mention one but the symbolic vocabulary
     only have the other.
* Due to a bug in the robot firmware, many kitchen and washing
    equipments can only take the binary state ON but not the binary
    state INSIDE. For example, the robot can understand {{"from_name
    ": "clothes_pants", "relation": "ON", "to_name": "washing_macine
    "}} but not {{"from_name": "clothes_pants", "relation": "INSIDE
    ", "to_name": "washing_machine"}}. Some equipments sustaining
    this bug are: washing_machine , dishwasher , oven , coffe_maker
    . The only equipment that can work with INSIDE is "freezer".
```

Snippet 2: VirtualHome Inconsistency Patching Rules

```
* Due to the space limit of the tape you can only output a few edge,
     node, and actions goal states overall. Only output the set of
    goal states that are directly relevant to the task name and the
    final part of the task description.
```

Snippet 3: Transition Modeling Algorithmic Strategy

```
The general strategy you might use is
1. Analyze the Problem (Parse Input)
2. Work Backward from the Goal (Define Effects). The easiest way to
     start is by looking at the :goal and mapping it to an action.
3. Define Preconditions (The "Why"). Now that you have an effect,
    ask: "What must be true for this action to happen?"
for examples: Obvious Preconditions: the state must be the opposite
     of the effect, the agent must be present, the agent must have
    the tool; Inferential Preconditions (The "Key Logic"): Look at
    the other actions. Why are there two cleaning actions (clean-
    dusty and clean-stained)? Find a predicate that captures this
    difference. The (soaked ?rag) predicate is the key.
4. Recurse on New Subgoals. You just created new subgoals (
    preconditions). Find the action that achieves this subgoal and
    repeat the same process.
5. Handle State Changes and Deletes (Rigor). Actions don't just add
     facts; they also delete them.
6. Assemble the Final Plan. Finally, trace the complete logical
    chain you have built, starting from the :init state. If the plan
     is not reachable, rework the action definitions again.
```

Snippet 4: VirtualHome Transition Modeling Prompt Snippet Emphasizing Common Action Usage

```
Here are some other commonly used actions and their PDDL
    definitions. Reuse these IN FULL whenever you are asked to
    define actions with these names. They are tried and true for
    the environment. When you are asked to define actions not in
    this set, make sure they are sensible and generalizable
    actions rather than overfitted actions designed specifically
    as a link toward the goals. For examples, the action `
    walk_towards` will need to have the full effect with both
    forall clauses, even if they are redundant for the problem
    asked.
```

D Prompt Templates

D.1 Goal Interpretation

D.1.1 BEHAVIOR

Template 1: BEHAVIOR – Goal Interpretation Original Template

```
You are a helpful assistant for goal interpretation in an embodied
    environment. You should only output in json format. Your task is
     to understand natural language goals for a household robot,
    reason about the object states and relationships, and turn
    natural language goals into symbolic goal states in the
    designated format. The goals include: unary goals describing one
     object's own unary states, and binary goals describing object-
    object binary relationships. The input will be the goal's name,
    the goal's description, relevant objects as well as their
    possible unary states, and all initial unary and binary states.
    The output should be the symbolic version of the goal states.


Relevant objects in the scene indicates those objects involved in
    the action execution initially. It will include the object name,
     and the object's all possible unary states (In goal conditions,
     each state can be set to true: ["Stained", "cup.n.01_1"] or not
     true: ["not", ["Stained", "cup.n.01_1"]]). It follows the
    format: object name including object id, possible unary states:
    ...(all possible unary states). Your proposed unary object
    states should be within the following set: {{"Cooked", "Open", "
    Frozen", "Dusty", "Stained", "Sliced", "Soaked", "Toggled_On"}}


Relevant objects in the scene are:
{object_in_scene}

All initial states in the scene are:
{all_initial_states}

Symbolic goals format:
```

```
Node goal states should be a set indicating the desired final goal
    states of single objects. Each goal in the list should be a list
     with two elements: the first element is the state name, which
    comes from the set {{"Cooked", "Open", "Frozen", "Dusty", "
    Stained", "Sliced", "Soaked", "Toggled_On"}}; the second element
     is the object name, which comes from the list of relevant
    objects in the scene provided above An example being ["Frozen",
    "chicken.n.01_1"]. To indicate the negation of a state such as
    the above example, simply use the format ["not", ["Frozen", "
    chicken.n.01_1"]].


Edge goal states should be a set indicating the desired binary
    relationships between two objects. Each goal state in the set is
     a list of three elements: the first element is the state name,
    which comes from the set {{"NextTo", "Inside", "OnFloor", "
    Touching", "Under"}}, the second and third elements are the
    object names, with relationship as indicated by the first
    element. For example, the edge goal condition ["inside", "tomato
    .n.01_2", "electric_refrigerator.n.01_1"] indicates that the
    object "tomato.n.01_2" is inside "electric_refrigerator.n.01_1",
     and the edge goal condition ["ontop", "plate.n.04_1", "
    countertop.n.01_1"] indicates that the object "plate.n.04_1" is
    ontop of the object "countertop.n.01_1". Similar to node goal
    states, edge goal states and also be negated by simply nesting
    inside a "not", for example ["not", ["inside", "peach.n.03_1", "
    jar.n.01_2"]] means that the object "peach.n.03_1" is not inside
     the object "jar.n.01_2".


Task Name and Goal Instructions:
{instructions_str}

Now using json format, output just the symbolic version of the goal
     states without any explanation. Output a single json object
    string, whose keys are 'node goals' and 'edge goals', and values
     are your output of symbolic node goals and symbolic edge goals,
     respectively. That is, your output should be of the format: {{"
    node goals": SYMBOLIC_NODE_GOALS, "edge goals":
    SYMBOLIC_EDGE_GOALS}}. Also, please strictly follow the
    aforementioned symbolic goal format.
```

Template 2: BEHAVIOR – Goal Interpretation Updated Template

```
You are an expert in symbolic AI who is translating natural
    language goals into a set of symbolic goal states for a
    household robot in an embodied environment. You should only
    output in json format. Your task is to understand natural
    language goals for a household robot, reason about the world
    model with object states, relationships, and translate natural
    language goals into symbolic goal states in the designated
    format. The goals include: unary goals describing one object's
    own unary states, and binary goals describing object-object
    binary relationships. The input will be the goal's name, the
    goal's description, relevant objects with their potential
```

```
    initial unary states, and all realized initial unary and binary
    states. The output should be the symbolic version of the goal
    states.

# SYMBOLIC REPRESENTATION FORMAT:

* Node states are unary state of objects. Each node state predicate
     is a list with two elements: the first element is the state
    name, which comes from the set {{"Cooked", "Burnt", "Dusty", "
    Frozen", "Open", "Sliced", "Soaked", "Stained", "ToggledOn", "
    Holds_Rh", "Holds_Lh"}}, the second element is the object name,
    which comes from the list of relevant objects in the scene. Each
     state predicate, for example ["Stained", "cup.n.01_1"],
    evaluates to true on its own, and its negated form by the format
     ["not", ["Stained", "cup.n.01_1"]] evaluates to false. An
    example being ["Frozen", "chicken.n.01_1"] means that the object
     "chicken.n.01_1" is frozen. After thawning, its state will
    transition into ["not", ["Frozen", "chicken.n.01_1"]].
* Edge states are binary state of objects. Each Edge state is a
    list of three elements: the first element is the state name,
    which comes from the set {{"OnTop", "NextTo", "Inside", "OnFloor
    ", "Touching", "Under"}}, the second and third elements are the
    object names, with relationship as indicated by the first
    element. For example, the edge state ["inside", "tomato.n.01_2",
     "electric_refrigerator.n.01_1"] indicates that the object "
    tomato.n.01_2" is inside "electric_refrigerator.n.01_1", and the
     edge state ["ontop", "plate.n.04_1", "countertop.n.01_1"]
    indicates that the object "plate.n.04_1" is ontop of the object
    "countertop.n.01_1". Each edge state predicate also evaluate to
    True on its own, and can also be negated by simply nesting
    inside a two-element list with "not" as the first element, for
    example ["not", ["inside", "peach.n.03_1", "jar.n.01_2"]] means
    that the object "peach.n.03_1" is not inside the object "jar.n
    .01_2".

Below is a list of available states and their descriptions.
| State Name | Arguments | Description |
| --- | --- | --- |
| inside | obj1.id, obj2.id | obj1 is inside obj2 . An agent cannot
     be inside anything |
| ontop | obj1.id, obj2.id | obj1 is on top of obj2 |
| nextto | obj1.id, obj2.id | obj1 is next to obj2 |
| under | obj1.id, obj2.id | obj1 is under obj2 |
| onfloor | obj1.id, floor2.id | obj1 is on the floor2 |
| touching | obj1.id, obj2.id | obj1 is next to and touching obj2 |
| cooked | obj1.id | obj1 is cooked |
| burnt | obj1.id | obj1 is burnt |
| dusty | obj1.id | obj1 is dusty |
| frozen | obj1.id | obj1 is frozen |
| open | obj1.id | obj1 is open |
| sliced | obj1.id | obj1 is sliced |
| soaked | obj1.id | obj1 is soaked |
| stained | obj1.id | obj1 is stained |
| toggledon | obj1.id | obj1 is toggled on |
| holds_rh | obj1.id | obj1 is in the right hand of the robot |
| holds_lh | obj1.id | obj1 is in the left hand of the robot |
```

```
Relevant objects in the scene indicates the potential involvement
    those objects in the sequence action executions required to
    achieve the final goal. Each object is listed in the form <
    object>: [possible unary states]. The actual intial states are
    listed as predicates along with binary states.

# INPUT FORMAT:
Relevant objects in the scene are:
<Objects in the scene with their possible unary states>

All initial states in the scene are:
<Realized initial states>

Task Name and Goal Instructions:
{{
   "Task Name": <task identifier>,
   "Goal Instructions": <natural language goal instruction>
}}

# OUTPUT FORMAT:
The output of node goals should be a list of node states represent
    the final goal states of each single object, according to the
    natural language goal instructions. Edge goals should be a list
    of edge states represent the final binary relationships between
    two objects, according to the natural language goal instructions
    . The output format must strictly follow the JSON format {{"node
     goals": [SYMBOLIC_NODE_GOAL_STATES], "edge goals": [
    SYMBOLIC_EDGE_GOAL_STATES]}}.

# EXAMPLE 1:
Below is an example for your better understanding of the problem
    and the input/output formats.

Relevant objects in the scene are:
grill.n.02_1: ['Stained', 'Dusty']
floor.n.01_1: ['Stained', 'Dusty']
rag.n.01_1: ['Stained', 'Soaked']
bucket.n.01_1: ['Stained', 'Dusty']
table.n.02_1: ['Stained', 'Dusty']
sink.n.01_1: ['ToggledOn', 'Stained', 'Dusty']

All initial states in the scene are:
['onfloor', 'grill.n.02_1', 'floor.n.01_1']
['stained', 'grill.n.02_1']
['dusty', 'grill.n.02_1']
['ontop', 'bucket.n.01_1', 'table.n.02_1']
['ontop', 'rag.n.01_1', 'table.n.02_1']
['onfloor', 'agent.n.01_1', 'floor.n.01_1']

Task Name and Goal Instructions:
{{
   "Task Name": "cleaning_barbecue_grill",
   "Goal Instructions": "Clean stains and dust off of the barbecue
       grill."
}}

Output:
```

```
{{"node goals": [["not", ["Stained", "grill.n.02_1"]], ["not", ["
    Dusty", "grill.n.02_1"]]], "edge goals": []}}


# EXAMPLE 2:
Below is an example for your better understanding of the problem
    and the input/output formats.

Relevant objects in the scene are:
electric_refrigerator.n.01_1: ['Stained', 'Open', 'Dusty']
food.n.01_1: ['Frozen']
food.n.01_2: ['Frozen']
food.n.01_3: ['Frozen']
cleansing_agent.n.01_1: ['Frozen']
table.n.02_1: ['Stained', 'Dusty']
towel.n.01_1: ['Stained', 'Soaked', 'Dusty']
floor.n.01_1: ['Stained', 'Dusty']
sink.n.01_1: ['ToggledOn', 'Stained', 'Dusty']
countertop.n.01_1: ['Stained', 'Dusty']
stove.n.01_1: ['ToggledOn', 'Stained', 'Dusty']
door.n.01_1: ['Stained', 'Open', 'Dusty']
chair.n.01_1: ['Stained', 'Dusty']

All initial states in the scene are:
['stained', 'electric_refrigerator.n.01_1']
['inside', 'food.n.01_1', 'electric_refrigerator.n.01_1']
['inside', 'food.n.01_2', 'electric_refrigerator.n.01_1']
['inside', 'food.n.01_3', 'electric_refrigerator.n.01_1']
['ontop', 'cleansing_agent.n.01_1', 'table.n.02_1']
['ontop', 'towel.n.01_1', 'table.n.02_1']
['onfloor', 'agent.n.01_1', 'floor.n.01_1']

Task Name and Goal Instructions:
{{
   "Task Name": "cleaning_freezer",
   "Goal Instructions": "Clean the stained freezer and remove all
       food items from the freezer."
}}

Output:
{{"node goals": [["not", ["Stained", "electric_refrigerator.n.01_1
    "]]], "edge goals": [["not", ["Inside", "food.n.01_1", "
    electric_refrigerator.n.01_1"]], ["not", ["Inside", "food.n.01_2
    ", "electric_refrigerator.n.01_1"]], ["not", ["Inside", "food.n
    .01_3", "electric_refrigerator.n.01_1"]]]}}


Now, it is your turn to translate the following natural language
    goal into symbolic goal states. The output needs no explanation,
     suffix, prefix. Strictly output the JSON format and nothing
    else.

Relevant objects in the scene are:
{object_in_scene}

All initial states in the scene are:
{all_initial_states}
```

```
Task Name and Goal Instructions:
{instructions_str}

Output:
```

D.1.2 VirtualHome

Template 3: VirtualHome – Goal Interpretation Original Template

```
Your task is to understand natural language goals for a household
    robot, reason about the object states and relationships, and
    turn natural language goals into symbolic goals in the given
    format. The goals include: node goals describing object states,
    edge goals describing object relationships and action goals
    describing must-to-do actions in this goal. The input will be
    the goal's name, the goal's description, relevant objects as
    well as their current and all possible states, and all possible
    relationships between objects. The output should be the symbolic
     version of the goals.


Relevant objects in the scene indicates those objects involved in
    the action execution initially. It will include the object name,
     the object initial states, and the object all possible states.
    It follows the format: object name, id: ...(object id), states:
    ...(object states), possible states: ...(all possible states).
    Your proposed object states should be within the following set:
    CLOSED, OPEN, ON, OFF, SITTING, DIRTY, CLEAN, LYING, PLUGGED_IN,
     PLUGGED_OUT.


Relevant objects in the scene are:
{object_in_scene}

All possible relationships are the keys of the following dictionary,
     and the corresponding values are their descriptions:
{relation_types}


Symbolic goals format:
Node goals should be a list indicating the desired ending states of
     objects. Each goal in the list should be a dictionary with two
    keys 'name' and 'state'. The value of 'name' is the name of the
    object, and the value of 'state' is the desired ending state of
    the target object. For example, [{{'name': 'washing_machine', '
    state': 'PLUGGED_IN'}}, {{'name': 'washing_machine', 'state': '
    CLOSED'}}, {{'name': 'washing_machine', 'state': 'ON'}}]
    requires the washing_machine to be PLUGGED_IN, CLOSED, and ON.
    It can be a valid interpretation of natural language goal:
Task name: Wash clothes.
Task description: Washing pants with washing machine
This is because if one wants to wash clothes, the washing machine
    should be functioning, and thus should be PLUGGED_IN, CLOSED,
    and ON.
```

```
Edge goals is a list of dictionaries indicating the desired
    relationships between objects. Each goal in the list is a
    dictionary with three keys 'from_name', and 'relation' and '
    to_name'. The value of 'relation' is desired relationship
    between 'from_name' object to 'to_name' object. The value of '
    from_name' and 'to_name' should be an object name. The value of
    'relation' should be an relationship. All relations should only
    be within the following set: ON, INSIDE, BETWEEN, CLOSE, FACING,
     HOLDS_RH, HOLDS_LH.

Each relation has a fixed set of objects to be its 'to_name' target.
     Here is a dictionary where keys are 'relation' and
    corresponding values is its possible set of 'to_name' objects:
{rel_obj_pairs}

Action goals is a list of actions that must be completed in the
    goals. The number of actions is less than three. If node goals
    and edge goals are not enough to fully describe the goal, add
    action goals to describe the goal. Below is a dictionary of
    possible actions, whose keys are all possible actions and values
     are corresponding descriptions. When output actions goal list,
    each action goal should be a dictionary with keys 'action' and '
    description'.
{action_space}

Goal name and goal description:
{goal_str}

Now output the symbolic version of the goal. Output in json format,
     whose keys are 'node goals', 'edge goals', and 'action goals',
    and values are your output of symbolic node goals, symbolic edge
     goals, and symbolic action goals, respectively. That is, {{'
    node goals': SYMBOLIC NODE GOALS, 'edge goals': SYMBOLIC EDGE
    GOALS, 'action goals': SYMBOLIC ACTION GOALS}}. Please strictly
    follow the symbolic goal format.
```

Template 4: VirtualHome – Goal Interpretation Updated Template (v1)

```
You are an expert in symbolic AI who is translating natural
    language goals into a set of symbolic goal states for an
    assistive household robot in an embodied environment. You should
     only output in json format. Your task is to understand natural
    language goals for a household robot, reason about the world
    model with object states, relationships, and translate natural
    language goals into symbolic goal states in the designated
    format. The goals include: unary goals describing one object's
    own unary states, and binary goals describing object-object
    binary relationships. The input will be the goal's name, the
    goal's description, relevant objects with their potential
    initial unary states, and all realized initial unary and binary
    states. The output should be the symbolic version of the goal
    states.

The robot you are working with maintains a tape of predicates to
```

```
    track the target the environment. Only predicates in this tape
    are evaluated to true. When a state is negated, it is removed
    from the list. Then, the robot will solve for plans to assist
    the primary agent (the character) to achieve these goal
    predicates. Since the sets of possible unary and binary
    relationships are limited, the final goal state can be more
    thoroughly described with action states of the character in some
     cases. However you cannot use more than three action states in
    the output which would break the robot. The need for action
    states are RARE and the robot have limited capability to deal
    with them, so you would only consider adding them to the output
    if the goal states cannot be fully described by unary and binary
     states and the action states added need to be an unambiguously
    persistent ongoing action of the character at the final goal
    state.

# SYMBOLIC REPRESENTATION FORMAT:
* Node states are unary states of objects. Each node state
    predicate is a dictionary with two keys 'name' and 'state'. The
    value of 'name' is the name of the object from a given list of
    relevant objects in the scene, and the value of 'state' is the
    name of the object's state, which must strictly come from the
    vocabulary {{ 'CLOSED', 'OPEN', 'ON', 'OFF', 'SITTING', 'DIRTY',
     'CLEAN', 'LYING', 'PLUGGED_IN', 'PLUGGED_OUT' }}. For example,
    {{'name': 'television', 'state': 'PLUGGED_IN'}} means the
    television is PLUGGED_IN, {{'name': 'washing_machine', 'state':
    'ON'}} means the washing machine is ON. Note that there are both
     a unary and a binary variants of SITTING.
* Edge states are binary state of objects. Each edge state is a
    dictionary with three keys 'from_name', 'relation', and 'to_name
    '. The value of 'relation' is desired relationship between '
    from_name' object to 'to_name' object. The values of 'from_name'
     and 'to_name' must be from a given list of relevant objects in
    the scene. The value of 'relation' is a primitive from the
    following set: {{ 'ON', 'INSIDE', 'BETWEEN', 'CLOSE', 'FACING',
    'HOLDS_RH', 'HOLDS_LH', SITTING }}, with the subset of actual
    relevant edge primitives given below. For example, {{'from_name
    ': character, 'relation': FACING, 'to_name': television}} means
    the character is FACING the television. Note that there are both
     a unary and a binary variants of SITTING.
* Action states are dynamic states of the character describing an
    ongoing action. Each action state is a dictionary with two keys
    'action' and 'description' of which the solely viable values are
     given below in a list of viable actions for the scene. For
    example, {{'action': 'LOOKAT', 'description': 'look at sth, face
     sth'}} describes the state of the character looking at
    something.

# SCENE DETAILS:
* Relevant objects in the scene indicates the potential involvement
     those objects in the sequence action executions required to
    achieve the final goal. Each object is listed on a line in the
    form
<object_name>, initial states: [<list of object initial unary
    states>], possible states: [<list of object admissible unary
    states>]
Note, however, that our current robot will not work with all the
```

```
    states in possible states list. You must use the overlap between
     elements of this list and our vocabulary {{ 'CLOSED', 'OPEN', '
    ON', 'OFF', 'SITTING', 'DIRTY', 'CLEAN', 'LYING', 'PLUGGED_IN',
    'PLUGGED_OUT' }} otherwise the robot will breakdown.
* Relevant edge primitives are the only viable values for the '
    relation' key in edge state predicates of this scene. You will
    be given a dictionary for the relevant edge primitives of form
    {{<edge primitive>: <description>}} All relevant edge primitives
     are the keys of the dictionary, and the corresponding values
    are their descriptions for your interpretation. Each of these
    binary edge primitives has a fixed set of objects to be its '
    to_name' values, which will be detailed in a dictionary of the
    form {{<edge primitive>: {{<set of possible 'to_name' values>}}
    }} after the dictionary of relevant edge primitives.
* Relevant action primitives are the only viable values for the '
    action' key and 'description' key in action state predicates of
    this scene. You will be given a dictionary for the relevant
    action primitives of form {{<action primitive>: <description>}}.
     All viable values for the 'action' key are the keys this
    dictionary, with the corresponding values being the values for '
    description' key in the output predicate.

# INPUT FORMAT
Relevant objects in the scene are:
<line-separated list of relevant objects>

Relevant edge primitives are:
<dictionary of relevant edge primitives>
Here is a dictionary of viable 'to_name' values for each edge
    primitive:
<dictionary of viable 'to_name' values for each edge primitive>

Relevant action primitives are:
<dictionary of relevant action primitives>

Goal name and goal description:
Goal name: <goal name>

Goal description: <goal description>

# OUTPUT FORMAT
You need to use the following JSON format to output for the robot,
    with a maximum of three action goals. Otherwise you will break
    an expensive robot.
{{
   'node_goals': [<list of node goals predicates>]
   'edge_goals': [<list of edge goals predicates>]
   'action_goals': [<list of action goals predicates>]
}}

# EXAMPLE
Below is an example for your better understanding.

Relevant objects in the scene are:
couch, initial states: ['CLEAN'], possible states: ['CLEAN', 'DIRTY
    ', 'FREE', 'OCCUPIED']
dining_room, initial states: ['CLEAN'], possible states: ['CLEAN']
```

```
home_office, initial states: ['CLEAN'], possible states: ['CLEAN']
television, initial states: ['OFF', 'CLEAN', 'PLUGGED_OUT'],
    possible states: ['CLEAN', 'OFF', 'ON', 'PLUGGED_IN', '
    PLUGGED_OUT', 'BROKEN']
remote_control, initial states: ['CLEAN'], possible states: ['CLEAN
    ', 'OFF', 'ON', 'GRABBED']
character, initial states: [], possible states: ['LYING', 'SITTING
    ']

Relevant edge primitives are:
{{'ON': 'An object rests atop another, like a book on a table.', '
    FACING': 'One object is oriented towards another, as in a person
     facing a wall.', 'HOLDS_LH': 'An object is held or supported by
     the left hand, like a left hand holding a ball.', 'INSIDE': 'An
     object is contained within another, like coins inside a jar.',
    'BETWEEN': 'An object is situated spatially between two entities
    , like a park between two buildings.', 'HOLDS_RH': 'An object is
     grasped or carried by the right hand, such as a right hand
    holding a pen.', 'CLOSE': 'Objects are near each other without
    touching, like two close-standing trees.'}}
Here is a dictionary of viable 'to_name' values for each edge
    primitive:
{{'ON': {{'bed', 'dishwasher', 'table', 'couch', 'oven', 'character
    ', 'toilet', 'washing_machine', 'coffe_maker'}}, 'HOLDS_LH': {{'
    keyboard', 'novel', 'tooth_paste', 'water_glass', 'toothbrush',
    'spectacles'}}, 'HOLDS_RH': {{'cup', 'address_book', '
    drinking_glass', 'remote_control', 'mouse', 'novel', '
    tooth_paste', 'water_glass', 'phone', 'toothbrush'}}, 'INSIDE':
    {{'home_office', 'bathroom', 'dining_room', 'hands_both', '
    freezer'}}, 'FACING': {{'television', 'computer', '
    remote_control', 'toilet', 'laptop', 'phone'}}, 'CLOSE': {{'
    shower', 'cat'}}}}

Relevant action primitives are:
{{'CLOSE': 'closing sth, meaning changing the state from unary OPEN
     to unary CLOSED', 'DRINK': 'drink up sth', 'FIND': 'find and
    get near to sth', 'WALK': 'walk towards sth, get near to sth', '
    GRAB': 'graph sth', 'LOOKAT': 'look at sth, face sth', '
    LOOKAT_SHORT': 'shortly look at sth', 'LOOKAT_LONG': 'look at
    sth for long', 'OPEN': 'open sth, as opposed to close sth', '
    POINTAT': 'point at sth', 'PUTBACK': 'put object A back to
    object B', 'PUTIN': 'put object A into object B', 'PUTOBJBACK':
    'put object back to its original place', 'RUN': 'run towards sth
    , get close to sth', 'SIT': 'sit on sth', 'STANDUP': 'stand up',
     'SWITCHOFF': 'switch sth off (normally lamp/light)', 'SWITCHON
    ': 'switch sth on (normally lamp/light)', 'TOUCH': 'touch sth',
    'TURNTO': 'turn and face sth', 'WATCH': 'watch sth', 'WIPE': '
    wipe sth out', 'PUTON': 'put on clothes, need to hold the
    clothes first', 'PUTOFF': 'put off clothes', 'GREET': 'greet to
    somebody', 'DROP': 'drop something in robot's current room, need
     to hold the thing first', 'READ': 'read something, need to hold
     the thing first', 'LIE': 'lie on something, need to get close
    the thing first', 'POUR': 'pour object A into object B', 'TYPE':
     'type on keyboard', 'PUSH': 'move sth', 'PULL': 'move sth', '
    MOVE': 'move sth', 'WASH': 'wash sth', 'RINSE': 'rinse sth', '
    SCRUB': 'scrub sth', 'SQUEEZE': 'squeeze the clothes', 'PLUGIN':
     'plug in the plug', 'PLUGOUT': 'plug out the plug', 'CUT': 'cut
```

```
     some food', 'EAT': 'eat some food', 'RELEASE': 'drop sth inside
     the current room'}}

Goal name and goal description:
Goal name: Watch TV

Goal description: I walk into the living room. I sit on the couch.
    I pick up the remote control. I push the power button. I push
    the guide button and look for my favorite show "The Middle." I
    click that channel and enjoy my show.

Output:
{{
  "node goals": [
    {{"name": "television", "state": "PLUGGED_IN"}},
    {{"name": "television", "state": "ON"}}
    {{"name": "character", "state": "SITTING"}}
  ],
  "edge goals": [
    {{"from_name": "character", "relation": "ON", "to_name": "couch
        "}}
    {{"from_name": "character", "relation": "HOLDS_RH", "to_name": "
        remote_control"}},
    {{"from_name": "character", "relation": "FACING", "to_name": "
        television"}}
  ],
  "action goals": [
    {{"action": "WATCH", "description": "watch sth"}}
  ]
}}

Explanation (not part of the output):
To translate this goal, we filtered the narrative to identify only
    the final, persistent states that define the act of "watching TV
    ," while ignoring the intermediate steps. The narrative's final
    phrase, "enjoy my show", directly corresponds to the goal name "
    Watch TV" and solidifies our choice of the ongoing WATCH action
    as the central purpose. We determined the rest of the final
    state by processing the key actions that lead to this. "Sit on
    the couch" defines the character's final position. We couldn't
    use a single binary SITTING relation because the binary SITTING
    primitive was not in this scene's specific list of allowed
    Relevant edge primitives. Therefore, we represented this concept
     by combining two states: the character's posture as a node
    state (SITTING) and their location as an edge state (ON couch).
    Similarly, "pick up the remote" results in the persistent
    HOLDS_RH remote_control state, and "push power" results in the
    television being ON. We also inferred that the TV must be
    powered on and operating, and thus must be PLUGGED_IN, and ON,
    and he character must be FACING the television as preconditions
    for the character to WATCH it. In contrast, we deliberately
    omitted actions like "walk...", "push the guide button", "look
    for...", and "click...". These are all transitional steps that
    describe the process of starting to watch, not the final,
    sustained "Watch TV" state itself.

---
```

```
Now it is your turn to translate the following task. Remember to
    strictly adhere to the lists of viable values in your answer. No
     explanation, prefix, or suffix is needed. Strictly use the
    OUTPUT FORMAT for your response and nothing else.

Relevant objects in the scene are:
{object_in_scene}

Relevant edge primitives are:
{relation_types}
Here is a dictionary of viable 'to_name' values for each edge
    primitive:
{rel_obj_pairs}

Relevant action primitives are:
{action_space}

Goal name and goal description:
{goal_str}

Output:
```

Template 5: VirtualHome – Goal Interpretation Updated Template (v2)

```
You are an expert in symbolic AI who is translating natural
    language goals into a set of symbolic goal states for an
    assistive household robot in an embodied environment. You should
     only output in json format. Your task is to understand natural
    language goals for a household robot, reason about the world
    model with object states, relationships, and translate natural
    language goals into symbolic goal states in the designated
    format. The goals include: unary goals describing one object's
    own unary states, and binary goals describing object-object
    binary relationships. The input will be the goal's name, the
    goal's description, relevant objects with their potential
    initial unary states, and all realized initial unary and binary
    states. The output should be the symbolic version of the goal
    states.

The robot you are working with maintains a tape of predicates to
    track the target the environment. Only predicates in this tape
    are evaluated to true. When a state is negated, it is removed
    from the list. Then, the robot will solve for plans to assist
    the primary agent (the character) to achieve these goal
    predicates. Since the sets of possible unary and binary
    relationships are limited, the final goal state can be more
    thoroughly described with action states of the character in some
     cases. However you cannot use more than three action states in
    the output which would break the robot. The need for action
    states are RARE and the robot have limited capability to deal
    with them, so you would only consider adding them to the output
    if the goal states cannot be fully described by unary and binary
     states and the action states added need to be an unambiguously
    persistent ongoing action of the character at the final goal
    state.
```

```
# SYMBOLIC REPRESENTATION FORMAT:
* Node states are unary states of objects. Each node state
    predicate is a dictionary with two keys 'name' and 'state'. The
    value of 'name' is the name of the object from a given list of
    relevant objects in the scene, and the value of 'state' is the
    name of the object's state, which must strictly come from the
    vocabulary {{ 'CLOSED', 'OPEN', 'ON', 'OFF', 'SITTING', 'DIRTY',
     'CLEAN', 'LYING', 'PLUGGED_IN', 'PLUGGED_OUT' }}. For example,
    {{'name': 'television', 'state': 'PLUGGED_IN'}} means the
    television is PLUGGED_IN, {{'name': 'washing_machine', 'state':
    'ON'}} means the washing machine is ON. Note that there are both
     a unary and a binary variants of SITTING.
* Edge states are binary state of objects. Each edge state is a
    dictionary with three keys 'from_name', 'relation', and 'to_name
    '. The value of 'relation' is desired relationship between '
    from_name' object to 'to_name' object. The values of 'from_name'
     and 'to_name' must be from a given list of relevant objects in
    the scene. The value of 'relation' is a primitive from the
    following set: {{ 'ON', 'INSIDE', 'BETWEEN', 'CLOSE', 'FACING',
    'HOLDS_RH', 'HOLDS_LH', SITTING }}, with the subset of actual
    relevant edge primitives given below. For example, {{'from_name
    ': character, 'relation': FACING, 'to_name': television}} means
    the character is FACING the television. Note that there are both
     a unary and a binary variants of SITTING.
* Action states are dynamic states of the character describing an
    ongoing action. Each action state is a dictionary with two keys
    'action' and 'description' of which the solely viable values are
     given below in a list of viable actions for the scene. For
    example, {{'action': 'LOOKAT', 'description': 'look at sth, face
     sth'}} describes the state of the character looking at
    something.

# SCENE DETAILS:
* Relevant objects in the scene indicates the potential involvement
     those objects in the sequence action executions required to
    achieve the final goal. Each object is listed on a line in the
    form
<object_name>, initial states: [<list of object initial unary
    states>], possible states: [<list of object admissible unary
    states>]
Note, however, that our current robot will not work with all the
    states in possible states list. You must use the overlap between
     elements of this list and our vocabulary {{ 'CLOSED', 'OPEN', '
    ON', 'OFF', 'SITTING', 'DIRTY', 'CLEAN', 'LYING', 'PLUGGED_IN',
    'PLUGGED_OUT' }} otherwise the robot will breakdown.
* Relevant edge primitives are the only viable values for the '
    relation' key in edge state predicates of this scene. You will
    be given a dictionary for the relevant edge primitives of form
    {{<edge primitive>: <description>}} All relevant edge primitives
     are the keys of the dictionary, and the corresponding values
    are their descriptions for your interpretation. Each of these
    binary edge primitives has a fixed set of objects to be its '
    to_name' values, which will be detailed in a dictionary of the
    form {{<edge primitive>: {{<set of possible 'to_name' values>}}
    }} after the dictionary of relevant edge primitives.
* Relevant action primitives are the only viable values for the '
```

```
    action' key and 'description' key in action state predicates of
    this scene. You will be given a dictionary for the relevant
    action primitives of form {{<action primitive>: <description>}}.
     All viable values for the 'action' key are the keys this
    dictionary, with the corresponding values being the values for '
    description' key in the output predicate.

# SPECIAL RULES
* Note that the "fridge" and the "freezer" often means the same
    thing to the robot and should be use interchangeably when the
    natural language command mention one but the symbolic vocabulary
     only have the other.
* Due to a bug in the robot firmware, many kitchen and washing
    equipments can only take the binary state ON but not the binary
    state INSIDE. For example, the robot can understand {{"from_name
    ": "clothes_pants", "relation": "ON", "to_name": "washing_macine
    "}} but not {{"from_name": "clothes_pants", "relation": "INSIDE
    ", "to_name": "washing_machine"}}. Some equipments sustaining
    this bug are: washing_machine , dishwasher , oven , coffe_maker
    . The only equipment that can work with INSIDE is "freezer".

# INPUT FORMAT
Relevant objects in the scene are:
<line-separated list of relevant objects>

Relevant edge primitives are:
<dictionary of relevant edge primitives>
Here is a dictionary of viable 'to_name' values for each edge
    primitive:
<dictionary of viable 'to_name' values for each edge primitive>

Relevant action primitives are:
<dictionary of relevant action primitives>

Goal name and goal description:
Goal name: <goal name>

Goal description: <goal description>

# OUTPUT FORMAT
You need to use the following JSON format to output for the robot,
    with a maximum of three action goals. Otherwise you will break
    an expensive robot.
{{
   'node_goals': [<list of node goals predicates>]
   'edge_goals': [<list of edge goals predicates>]
   'action_goals': [<list of action goals predicates>]
}}

# EXAMPLE
Below is an example for your better understanding.

Relevant objects in the scene are:
couch, initial states: ['CLEAN'], possible states: ['CLEAN', 'DIRTY
    ', 'FREE', 'OCCUPIED']
dining_room, initial states: ['CLEAN'], possible states: ['CLEAN']
home_office, initial states: ['CLEAN'], possible states: ['CLEAN']
```

```
television, initial states: ['OFF', 'CLEAN', 'PLUGGED_OUT'],
    possible states: ['CLEAN', 'OFF', 'ON', 'PLUGGED_IN', '
    PLUGGED_OUT', 'BROKEN']
remote_control, initial states: ['CLEAN'], possible states: ['CLEAN
    ', 'OFF', 'ON', 'GRABBED']
character, initial states: [], possible states: ['LYING', 'SITTING
    ']

Relevant edge primitives are:
{{'ON': 'An object rests atop another, like a book on a table.', '
    FACING': 'One object is oriented towards another, as in a person
     facing a wall.', 'HOLDS_LH': 'An object is held or supported by
     the left hand, like a left hand holding a ball.', 'INSIDE': 'An
     object is contained within another, like coins inside a jar.',
    'BETWEEN': 'An object is situated spatially between two entities
    , like a park between two buildings.', 'HOLDS_RH': 'An object is
     grasped or carried by the right hand, such as a right hand
    holding a pen.', 'CLOSE': 'Objects are near each other without
    touching, like two close-standing trees.'}}
Here is a dictionary of viable 'to_name' values for each edge
    primitive:
{{'ON': {{'bed', 'dishwasher', 'table', 'couch', 'oven', 'character
    ', 'toilet', 'washing_machine', 'coffe_maker'}}, 'HOLDS_LH': {{'
    keyboard', 'novel', 'tooth_paste', 'water_glass', 'toothbrush',
    'spectacles'}}, 'HOLDS_RH': {{'cup', 'address_book', '
    drinking_glass', 'remote_control', 'mouse', 'novel', '
    tooth_paste', 'water_glass', 'phone', 'toothbrush'}}, 'INSIDE':
    {{'home_office', 'bathroom', 'dining_room', 'hands_both', '
    freezer'}}, 'FACING': {{'television', 'computer', '
    remote_control', 'toilet', 'laptop', 'phone'}}, 'CLOSE': {{'
    shower', 'cat'}}}}

Relevant action primitives are:
{{'CLOSE': 'closing sth, meaning changing the state from unary OPEN
     to unary CLOSED', 'DRINK': 'drink up sth', 'FIND': 'find and
    get near to sth', 'WALK': 'walk towards sth, get near to sth', '
    GRAB': 'graph sth', 'LOOKAT': 'look at sth, face sth', '
    LOOKAT_SHORT': 'shortly look at sth', 'LOOKAT_LONG': 'look at
    sth for long', 'OPEN': 'open sth, as opposed to close sth', '
    POINTAT': 'point at sth', 'PUTBACK': 'put object A back to
    object B', 'PUTIN': 'put object A into object B', 'PUTOBJBACK':
    'put object back to its original place', 'RUN': 'run towards sth
    , get close to sth', 'SIT': 'sit on sth', 'STANDUP': 'stand up',
     'SWITCHOFF': 'switch sth off (normally lamp/light)', 'SWITCHON
    ': 'switch sth on (normally lamp/light)', 'TOUCH': 'touch sth',
    'TURNTO': 'turn and face sth', 'WATCH': 'watch sth', 'WIPE': '
    wipe sth out', 'PUTON': 'put on clothes, need to hold the
    clothes first', 'PUTOFF': 'put off clothes', 'GREET': 'greet to
    somebody', 'DROP': 'drop something in robot's current room, need
     to hold the thing first', 'READ': 'read something, need to hold
     the thing first', 'LIE': 'lie on something, need to get close
    the thing first', 'POUR': 'pour object A into object B', 'TYPE':
     'type on keyboard', 'PUSH': 'move sth', 'PULL': 'move sth', '
    MOVE': 'move sth', 'WASH': 'wash sth', 'RINSE': 'rinse sth', '
    SCRUB': 'scrub sth', 'SQUEEZE': 'squeeze the clothes', 'PLUGIN':
     'plug in the plug', 'PLUGOUT': 'plug out the plug', 'CUT': 'cut
     some food', 'EAT': 'eat some food', 'RELEASE': 'drop sth inside
```

```
     the current room'}}

Goal name and goal description:
Goal name: Watch TV

Goal description: I walk into the living room. I sit on the couch.
    I pick up the remote control. I push the power button. I push
    the guide button and look for my favorite show "The Middle." I
    click that channel and enjoy my show.

Output:
{{
  "node goals": [
    {{"name": "television", "state": "PLUGGED_IN"}},
    {{"name": "television", "state": "ON"}}
    {{"name": "character", "state": "SITTING"}}
  ],
  "edge goals": [
    {{"from_name": "character", "relation": "ON", "to_name": "couch
        "}}
    {{"from_name": "character", "relation": "HOLDS_RH", "to_name": "
        remote_control"}},
    {{"from_name": "character", "relation": "FACING", "to_name": "
        television"}}
  ],
  "action goals": [
    {{"action": "WATCH", "description": "watch sth"}}
  ]
}}

Explanation (not part of the output):
To translate this goal, we filtered the narrative to identify only
    the final, persistent states that define the act of "watching TV
    ," while ignoring the intermediate steps. The narrative's final
    phrase, "enjoy my show", directly corresponds to the goal name "
    Watch TV" and solidifies our choice of the ongoing WATCH action
    as the central purpose. We determined the rest of the final
    state by processing the key actions that lead to this. "Sit on
    the couch" defines the character's final position. We couldn't
    use a single binary SITTING relation because the binary SITTING
    primitive was not in this scene's specific list of allowed
    Relevant edge primitives. Therefore, we represented this concept
     by combining two states: the character's posture as a node
    state (SITTING) and their location as an edge state (ON couch).
    Similarly, "pick up the remote" results in the persistent
    HOLDS_RH remote_control state, and "push power" results in the
    television being ON. We also inferred that the TV must be
    powered on and operating, and thus must be PLUGGED_IN, and ON,
    and he character must be FACING the television as preconditions
    for the character to WATCH it. In contrast, we deliberately
    omitted actions like "walk...", "push the guide button", "look
    for...", and "click...". These are all transitional steps that
    describe the process of starting to watch, not the final,
    sustained "Watch TV" state itself.

---
Now it is your turn to translate the following task. Remember to
```

```
    strictly adhere to the lists of viable values in your answer. No
     explanation, prefix, or suffix is needed. Strictly use the
    OUTPUT FORMAT for your response and nothing else.

Relevant objects in the scene are:
{object_in_scene}

Relevant edge primitives are:
{relation_types}
Here is a dictionary of viable 'to_name' values for each edge
    primitive:
{rel_obj_pairs}

Relevant action primitives are:
{action_space}

Goal name and goal description:
{goal_str}

Output:
```

Template 6: VirtualHome – Goal Interpretation Updated Template (v3)

```
You are an expert in symbolic AI who is translating natural
    language goals into a set of symbolic goal states for an
    assistive household robot in an embodied environment. You should
     only output in json format. Your task is to understand natural
    language goals for a household robot, reason about the world
    model with object states, relationships, and translate natural
    language goals into symbolic goal states in the designated
    format. The goals include: unary goals describing one object's
    own unary states, and binary goals describing object-object
    binary relationships. The input will be the goal's name, the
    goal's description, relevant objects with their potential
    initial unary states, and all realized initial unary and binary
    states. The output should be the symbolic version of the goal
    states.

The robot you are working with maintains a tape of predicates to
    track the target the environment. Only predicates in this tape
    are evaluated to true. When a state is negated, it is removed
    from the list. Then, the robot will solve for plans to assist
    the primary agent (the character) to achieve these goal
    predicates. Since the sets of possible unary and binary
    relationships are limited, the final goal state can be more
    thoroughly described with action states of the character in some
     cases. However you cannot use more than three action states in
    the output which would break the robot. The need for action
    states are RARE and the robot have limited capability to deal
    with them, so you would only consider adding them to the output
    if the goal states cannot be fully described by unary and binary
     states and the action states added need to be an unambiguously
    persistent ongoing action of the character at the final goal
    state.
```

```
# SYMBOLIC REPRESENTATION FORMAT:
* Node states are unary states of objects. Each node state
    predicate is a dictionary with two keys 'name' and 'state'. The
    value of 'name' is the name of the object from a given list of
    relevant objects in the scene, and the value of 'state' is the
    name of the object's state, which must strictly come from the
    vocabulary {{ 'CLOSED', 'OPEN', 'ON', 'OFF', 'SITTING', 'DIRTY',
     'CLEAN', 'LYING', 'PLUGGED_IN', 'PLUGGED_OUT' }}. For example,
    {{'name': 'television', 'state': 'PLUGGED_IN'}} means the
    television is PLUGGED_IN, {{'name': 'washing_machine', 'state':
    'ON'}} means the washing machine is ON. Note that there are both
     a unary and a binary variants of SITTING.
* Edge states are binary state of objects. Each edge state is a
    dictionary with three keys 'from_name', 'relation', and 'to_name
    '. The value of 'relation' is desired relationship between '
    from_name' object to 'to_name' object. The values of 'from_name'
     and 'to_name' must be from a given list of relevant objects in
    the scene. The value of 'relation' is a primitive from the
    following set: {{ 'ON', 'INSIDE', 'BETWEEN', 'CLOSE', 'FACING',
    'HOLDS_RH', 'HOLDS_LH', SITTING }}, with the subset of actual
    relevant edge primitives given below. For example, {{'from_name
    ': character, 'relation': FACING, 'to_name': television}} means
    the character is FACING the television. Note that there are both
     a unary and a binary variants of SITTING.
* Action states are dynamic states of the character describing an
    ongoing action. Each action state is a dictionary with two keys
    'action' and 'description' of which the solely viable values are
     given below in a list of viable actions for the scene. For
    example, {{'action': 'LOOKAT', 'description': 'look at sth, face
     sth'}} describes the state of the character looking at
    something.

# SCENE DETAILS:
* Relevant objects in the scene indicates the potential involvement
     those objects in the sequence action executions required to
    achieve the final goal. Each object is listed on a line in the
    form
<object_name>, initial states: [<list of object initial unary
    states>], possible states: [<list of object admissible unary
    states>]
Note, however, that our current robot will not work with all the
    states in possible states list. You must use the overlap between
     elements of this list and our vocabulary {{ 'CLOSED', 'OPEN', '
    ON', 'OFF', 'SITTING', 'DIRTY', 'CLEAN', 'LYING', 'PLUGGED_IN',
    'PLUGGED_OUT' }} otherwise the robot will breakdown.
* Relevant edge primitives are the only viable values for the '
    relation' key in edge state predicates of this scene. You will
    be given a dictionary for the relevant edge primitives of form
    {{<edge primitive>: <description>}} All relevant edge primitives
     are the keys of the dictionary, and the corresponding values
    are their descriptions for your interpretation. Each of these
    binary edge primitives has a fixed set of objects to be its '
    to_name' values, which will be detailed in a dictionary of the
    form {{<edge primitive>: {{<set of possible 'to_name' values>}}
    }} after the dictionary of relevant edge primitives.
* Relevant action primitives are the only viable values for the '
    action' key and 'description' key in action state predicates of
```

```
    this scene. You will be given a dictionary for the relevant
    action primitives of form {{<action primitive>: <description>}}.
     All viable values for the 'action' key are the keys this
    dictionary, with the corresponding values being the values for '
    description' key in the output predicate.

# SPECIAL RULES
* Note that the "fridge" and the "freezer" often means the same
    thing to the robot and should be use interchangeably when the
    natural language command mention one but the symbolic vocabulary
     only have the other.
* Due to a bug in the robot firmware, many kitchen and washing
    equipments can only take the binary state ON but not the binary
    state INSIDE. For example, the robot can understand {{"from_name
    ": "clothes_pants", "relation": "ON", "to_name": "washing_macine
    "}} but not {{"from_name": "clothes_pants", "relation": "INSIDE
    ", "to_name": "washing_machine"}}. Some equipments sustaining
    this bug are: washing_machine , dishwasher , oven , coffe_maker
    . The only equipment that can work with INSIDE is "freezer".
* Due to the space limit of the tape you can only output a few edge,
     node, and actions goal states overall. Only output the set of
    goal states that are directly relevant to the task name and the
    final part of the task description.

# INPUT FORMAT
Relevant objects in the scene are:
<line-separated list of relevant objects>

Relevant edge primitives are:
<dictionary of relevant edge primitives>
Here is a dictionary of viable 'to_name' values for each edge
    primitive:
<dictionary of viable 'to_name' values for each edge primitive>

Relevant action primitives are:
<dictionary of relevant action primitives>

Goal name and goal description:
Goal name: <goal name>

Goal description: <goal description>

# OUTPUT FORMAT
You need to use the following JSON format to output for the robot,
    with a maximum of three action goals. Otherwise you will break
    an expensive robot.
{{
   'node_goals': [<list of node goals predicates>]
   'edge_goals': [<list of edge goals predicates>]
   'action_goals': [<list of action goals predicates>]
}}

# EXAMPLE
Below is an example for your better understanding.

Relevant objects in the scene are:
couch, initial states: ['CLEAN'], possible states: ['CLEAN', 'DIRTY
```

```
    ', 'FREE', 'OCCUPIED']
dining_room, initial states: ['CLEAN'], possible states: ['CLEAN']
home_office, initial states: ['CLEAN'], possible states: ['CLEAN']
television, initial states: ['OFF', 'CLEAN', 'PLUGGED_OUT'],
    possible states: ['CLEAN', 'OFF', 'ON', 'PLUGGED_IN', '
    PLUGGED_OUT', 'BROKEN']
remote_control, initial states: ['CLEAN'], possible states: ['CLEAN
    ', 'OFF', 'ON', 'GRABBED']
character, initial states: [], possible states: ['LYING', 'SITTING
    ']

Relevant edge primitives are:
{{'ON': 'An object rests atop another, like a book on a table.', '
    FACING': 'One object is oriented towards another, as in a person
     facing a wall.', 'HOLDS_LH': 'An object is held or supported by
     the left hand, like a left hand holding a ball.', 'INSIDE': 'An
     object is contained within another, like coins inside a jar.',
    'BETWEEN': 'An object is situated spatially between two entities
    , like a park between two buildings.', 'HOLDS_RH': 'An object is
     grasped or carried by the right hand, such as a right hand
    holding a pen.', 'CLOSE': 'Objects are near each other without
    touching, like two close-standing trees.'}}
Here is a dictionary of viable 'to_name' values for each edge
    primitive:
{{'ON': {{'bed', 'dishwasher', 'table', 'couch', 'oven', 'character
    ', 'toilet', 'washing_machine', 'coffe_maker'}}, 'HOLDS_LH': {{'
    keyboard', 'novel', 'tooth_paste', 'water_glass', 'toothbrush',
    'spectacles'}}, 'HOLDS_RH': {{'cup', 'address_book', '
    drinking_glass', 'remote_control', 'mouse', 'novel', '
    tooth_paste', 'water_glass', 'phone', 'toothbrush'}}, 'INSIDE':
    {{'home_office', 'bathroom', 'dining_room', 'hands_both', '
    freezer'}}, 'FACING': {{'television', 'computer', '
    remote_control', 'toilet', 'laptop', 'phone'}}, 'CLOSE': {{'
    shower', 'cat'}}}}

Relevant action primitives are:
{{'CLOSE': 'closing sth, meaning changing the state from unary OPEN
     to unary CLOSED', 'DRINK': 'drink up sth', 'FIND': 'find and
    get near to sth', 'WALK': 'walk towards sth, get near to sth', '
    GRAB': 'graph sth', 'LOOKAT': 'look at sth, face sth', '
    LOOKAT_SHORT': 'shortly look at sth', 'LOOKAT_LONG': 'look at
    sth for long', 'OPEN': 'open sth, as opposed to close sth', '
    POINTAT': 'point at sth', 'PUTBACK': 'put object A back to
    object B', 'PUTIN': 'put object A into object B', 'PUTOBJBACK':
    'put object back to its original place', 'RUN': 'run towards sth
    , get close to sth', 'SIT': 'sit on sth', 'STANDUP': 'stand up',
     'SWITCHOFF': 'switch sth off (normally lamp/light)', 'SWITCHON
    ': 'switch sth on (normally lamp/light)', 'TOUCH': 'touch sth',
    'TURNTO': 'turn and face sth', 'WATCH': 'watch sth', 'WIPE': '
    wipe sth out', 'PUTON': 'put on clothes, need to hold the
    clothes first', 'PUTOFF': 'put off clothes', 'GREET': 'greet to
    somebody', 'DROP': 'drop something in robot's current room, need
     to hold the thing first', 'READ': 'read something, need to hold
     the thing first', 'LIE': 'lie on something, need to get close
    the thing first', 'POUR': 'pour object A into object B', 'TYPE':
     'type on keyboard', 'PUSH': 'move sth', 'PULL': 'move sth', '
    MOVE': 'move sth', 'WASH': 'wash sth', 'RINSE': 'rinse sth', '
```

```
    SCRUB': 'scrub sth', 'SQUEEZE': 'squeeze the clothes', 'PLUGIN':
     'plug in the plug', 'PLUGOUT': 'plug out the plug', 'CUT': 'cut
     some food', 'EAT': 'eat some food', 'RELEASE': 'drop sth inside
     the current room'}}

Goal name and goal description:
Goal name: Watch TV

Goal description: I walk into the living room. I sit on the couch.
    I pick up the remote control. I push the power button. I push
    the guide button and look for my favorite show "The Middle." I
    click that channel and enjoy my show.

Output:
{{
  "node goals": [
    {{"name": "television", "state": "ON"}}
  ],
  "edge goals": [
    {{"from_name": "character", "relation": "FACING", "to_name": "
        television"}}
  ],
  "action goals": [
    {{"action": "WATCH", "description": "watch sth"}}
  ]
}}

Explanation (not part of the output):
To translate this goal, we filtered the narrative to identify only
    the final, persistent states that define the act of "watching TV
    ," while ignoring the intermediate steps. The narrative's final
    phrase, "enjoy my show", directly corresponds to the goal name "
    Watch TV" and solidifies our choice of the ongoing WATCH action
    as the central purpose. We determined the rest of the final
    state by processing the key actions that lead to this. "Sit on
    the couch" defines the character's final position. We couldn't
    use a single binary SITTING relation because the binary SITTING
    primitive was not in this scene's specific list of allowed
    Relevant edge primitives. Therefore, we represented this concept
     by combining two states: the character's posture as a node
    state (SITTING) and their location as an edge state (ON couch).
    Similarly, "pick up the remote" results in the persistent
    HOLDS_RH remote_control state, and "push power" results in the
    television being ON. We also inferred that the TV must be
    powered on and operating, and thus must be PLUGGED_IN, and ON,
    and he character must be FACING the television as preconditions
    for the character to WATCH it. In contrast, we deliberately
    omitted actions like "walk...", "push the guide button", "look
    for...", and "click...". These are all transitional steps that
    describe the process of starting to watch, not the final,
    sustained "Watch TV" state itself. Although there are multiple
    subgoals, we only keep the final states for each object in the
    scene as goal states. Note that if the task name was "Change TV
    channel" instead of "Watch TV", the final action goal would be
    PUSH or TOUCH the remote control, so the task name is very
    important for inference.
---
```

```
Now it is your turn to translate the following task. Remember to
    strictly adhere to the lists of viable values in your answer. No
     explanation, prefix, or suffix is needed. Strictly use the
    OUTPUT FORMAT for your response and nothing else.

Relevant objects in the scene are:
{object_in_scene}

Relevant edge primitives are:
{relation_types}
Here is a dictionary of viable 'to_name' values for each edge
    primitive:
{rel_obj_pairs}

Relevant action primitives are:
{action_space}

Goal name and goal description:
{goal_str}

Output:
```

D.2 Subgoal Decomposition

D.2.1 BEHAVIOR

Template 7: BEHAVIOR – Subgoal Decomposition Original Template

```
# Background Introduction
You are determining the complete state transitions of a household
    task solving by a robot. The goal is to list all intermediate
    states (which is called subgoals as a whole) to achieve the
    final goal states from the initial states. The output consists
    of a list of boolean expression, which is a combination of the
    state predicates. Note that your output boolean expression list
    is in temporal order, therefore, it must be consistent and
    logical. In short, your task is to output the subgoal plan in
    the required format.

# Data Vocabulary Introduction
Below we introduce the detailed data vocabulary and their format
    that you can use to generate the subgoal plan.
## Available States
The state is represented as a first-order predicate, which is a
    tuple of a predicate name and its arguments. Its formal
    definition looks like this "<PredicateName>(Params)", where <
    PredicateName> is the state name and each param should be ended
    with an id. An example is "inside(coin.1, jar.1)". Below is a
    list of available states and their descriptions.
| State Name | Arguments | Description |
| --- | --- | --- |
| inside | (obj1.id, obj2.id) | obj1 is inside obj2. If we have
    state inside(A, B), and you want to take A out of B while B is
    openable and stayed at "not open" state, please open B first.
    Also, inside(obj1, agent) is invalid.|
```

```
| ontop | (obj1.id, obj2.id) | obj1 is on top of obj2 |
| nextto | (obj1.id, obj2.id) | obj1 is next to obj2 |
| under | (obj1.id, obj2.id) | obj1 is under obj2 |
| onfloor | (obj1.id, floor2.id) | obj1 is on the floor2 |
| touching | (obj1.id, obj2.id) | obj1 is touching or next to obj2
    |
| cooked | (obj1.id) | obj1 is cooked |
| burnt | (obj1.id) | obj1 is burnt |
| dusty | (obj1.id) | obj1 is dusty. If want to change dusty(obj1.
    id) to "not dusty(obj1.id)", there are two ways to do it,
    depending on task conditions. Here, all objects other than obj1
    are types but not instances: 1. [inside(obj1.id, dishwasher or
    sink), toggledon(dishwasher or sink)] 2. holding other cleanning
     tool |
| frozen | (obj1.id) | obj1 is frozen |
| open | (obj1.id) | obj1 is open |
| sliced | (obj1.id) | obj1 is sliced. If want to change "not
    sliced(obj1.id)" to "sliced(obj1.id)", one must have a slicer. |
| soaked | (obj1.id) | obj1 is soaked |
| stained | (obj1.id) | obj1 is stained. If want to change stained(
    obj1.id) to "not stained(obj1.id)", there are three ways to do
    it, depending on task conditions. Here, all objects other than
    obj1 are types but not instances: 1. [inside(obj1.id, sink),
    toggledon(sink)] 2. [soaked(cleaner)] 3. holding detergent. |
| toggledon | (obj1.id) | obj1 is toggled on |
| holds_rh | (obj1.id) | obj1 is in the right hand of the robot |
| holds_lh | (obj1.id) | obj1 is in the left hand of the robot |
## Available Connectives
The connectives are used to satisfy the complex conditions. They
    are used to combine the state predicates.
| Connective Name | Arguments | Description |
| --- | --- | --- |
| and | exp1 and exp2 | evaluates to true if both exp1 and exp2 are
     true |
| or | exp1 or exp2 | evaluates to true if either exp1 or exp2 is
    true |
| not | not exp | evaluates to true if exp is false |
| forall | forall(x, exp) | evaluates to true if exp is true for
    all x |
| exists | exists(x, exp) | evaluates to true if exp is true for at
     least one x |
| forpairs | forpairs(x, y, exp) | evaluates to true if exp is true
     for all pairs of x and y. For example, forpairs(watch, basket,
    inside(watch, basket)) means that for each watch and basket, the
     watch is inside the basket. |
| forn | forn(n, x, exp) | evaluates to true if exp is true for
    exactly n times for x. For example, forn(2, jar_n_01, (not open(
    jar_n_01)) means that there are exactly two jars that are not
    open. |
| fornpairs | fornpairs(n, x, y, exp) | evaluates to true if exp is
     true for exactly n times for pairs of x and y. For example,
    fornpairs(2, watch, basket, inside(watch, basket)) means that
    there are exactly two watches inside the basket. |

# Rules You Must Follow
- The initial states are the states that are given at the beginning
     of the task.
```

```
- Your output must be a list of boolean expressions that are in
    temporal order.
- You must follow the data format and the available states and
    connectives defined above.
- The output must be consistent, logical and as detailed as
    possible. View your output as a complete state transition from
    the initial states to the final goal states.

- Please note that the robot can only hold one object in one hand.
    Also, the robot needs to have at least one hand free to perform
    any action other than put, place, take, hold.
- Use holds_rh and holds_lh in your plan if necessary. For example,
     stained(shoe) cannot directly change to not stained(shoe), but
    needs intermediate states like [soaked(rag), holds_rh(rag), not
    stained(shoe)] or [holds_rh(detergent), not stained(shoe)].
- Your output follows the temporal order line by line. If you think
     there is no temporal order requirement for certain states, you
    can use connective 'and' to combine them. If you think some
    states are equivalent, you can use connective 'or' to combine
    them.
- Please use provided relevant objects well to help you achieve the
     final goal states. Note that inside(obj1, agent) is an invalid
    state, therefore you cannot output it in your plan.
- Do not output redundant states. A redundant state means a state
    that is either not necessary or has been satisfied before
    without broken.
- Your must strictly follow the json format like this {"output": [<
    your subgoal plan>]}, where <your subgoal plan> is a list of
    boolean expressions presented in the temporal order.
- Start your output with "{" and end with "}". For each line of the
     output, DO NOT INCLUDE IRRELEVANT INFORMATION (like number of
    line, explanation, etc.).

# Example: Task is bottling_fruit
Below we provide an example for your better understanding.
## Relevant objects in this scene
{'name': 'strawberry.0', 'category': 'strawberry_n_01'}
{'name': 'fridge.97', 'category': 'electric_refrigerator_n_01'}
{'name': 'peach.0', 'category': 'peach_n_03'}
{'name': 'countertop.84', 'category': 'countertop_n_01'}
{'name': 'jar.0', 'category': 'jar_n_01'}
{'name': 'jar.1', 'category': 'jar_n_01'}
{'name': 'carving_knife.0', 'category': 'carving_knife_n_01'}
{'name': 'bottom_cabinet_no_top.80', 'category': 'cabinet_n_01'}
{'name': 'room_floor_kitchen.0', 'category': 'floor_n_01'}

## Initial States
inside(strawberry.0, fridge.97)
inside(peach.0, fridge.97)
not sliced(strawberry.0)
not sliced(peach.0)
ontop(jar.0, countertop.84)
ontop(jar.1, countertop.84)
ontop(carving_knife.0, countertop.84)
onfloor(agent_n_01.1, room_floor_kitchen.0)

## Goal States
```

```
exists(jar_n_01, (inside(strawberry.0, jar_n_01) and (not inside(
    peach.0, jar_n_01))))
exists(jar_n_01, (inside(peach.0, jar_n_01) and (not inside(
    strawberry.0, jar_n_01))))
forall(jar_n_01, (not open(jar_n_01)))
sliced(strawberry.0)
sliced(peach.0)

## Output: Based on initial states in this task, achieve final goal
     states logically and reasonably. It does not matter which state
     should be satisfied first, as long as all goal states can be
    satisfied at the end. Make sure your output follows the json
    format, and do not include irrelevant information, do not
    include any explanation. Output concrete states and do not use
    quantifiers like "forall" or "exists".
{"output": ["ontop(strawberry.0, countertop.84) and ontop(peach.0,
    countertop.84)", "holds_rh(carving_knife.0)", "sliced(strawberry
    .0) and sliced(peach.0)", "inside(strawberry.0, jar.0) and not
    inside(peach.0, jar.0)", "inside(peach.0, jar.1) and not inside(
    strawberry.0, jar.1)", "not open(jar.0) and not open(jar.1)"]}

Now, it is time for you to generate the subgoal plan for the
    following task.
# Target Task: {task_name}
## Relevant objects in this scene
{relevant_objects}

## Initial States
{initial_states}

## Goal States
{goal_states}

## Output: Based on initial states in this task, achieve final goal
     states logically and reasonably. It does not matter which state
     should be satisfied first, as long as all goal states can be
    satisfied at the end. Make sure your output follows the json
    format, and do not include irrelevant information, do not
    include any explanation. Output concrete states and do not use
    quantifiers like "forall" or "exists".

Now, it is time for you to generate the subgoal plan for the
    following task.
# Target Task: {task_name}
## Relevant objects in this scene
{relevant_objects}

## Initial States
{initial_states}

## Goal States
{goal_states}

## Output: Based on initial states in this task, achieve final goal
     states logically and reasonably. It does not matter which state
     should be satisfied first, as long as all goal states can be
    satisfied at the end. Make sure your output follows the json
```

```
    format, and do not include irrelevant information, do not
    include any explanation. Output concrete states and do not use
    quantifiers like "forall" or "exists".
```

Template 8: BEHAVIOR – Subgoal Decomposition Updated Template (v1)

```
# Background Introduction
You are an expert in symbolic AI who is determining the complete
    state transitions of a household task solving by a robot. The
    objective is to list all intermediate goal states (which are
    called subgoals as a whole) to achieve the final target goals.
    The state primitives include: unary primitives describing one
    object's own unary states and binary primitives describing
    object-object relationships. The output is a list of state and
    relation predicates constructed from the primitives detailed
    below.

More specifically, your task is to read and comprehend the input,
    reason about the world model with object states and
    relationships, and relevant (pseudo-)actions for state
    transitions required to traverse the list of subgoals to achieve
     the goals in terms of symbolic predicates. The robot's planner
    will take your output and attempt to plan for action sequences
    to iteratively achieve each subgoal. Thus, your list of subgoals
     needs to follow temporal logic order so success completions of
    subgoals can lead to achieving the final goal states.

# State Primitives Vocabulary
Below we introduce the detailed data vocabulary and their format
    that you can use to generate the subgoal plan.
## Available States
The state primitive is a tuple of a predicate name and its
    arguments. Its formal definition looks like this "<PredicateName
    >(Params)", where <PredicateName> is the state name and each
    param should be ended with an id. For example, if a jar is
    opened, it is represented as "opened(jar.1)". Another example is
     "inside(coin.1, jar.1)" which describes coin.1 is inside jar.1.
     Below is a list of available states and their descriptions.
| State Name | Arguments | Description |
| --- | --- | --- |
| inside | (obj1.id, obj2.id) | obj1 is inside obj2. agent cannot
    be an argument of this state. If we have state inside(A, B), and
     you want to take A out of B you need to reach open(B) by
    openning B first |
| ontop | (obj1.id, obj2.id) | obj1 is on top of obj2 |
| nextto | (obj1.id, obj2.id) | obj1 is next to obj2 |
| under | (obj1.id, obj2.id) | obj1 is under obj2 |
| onfloor | (obj1.id, floor2.id) | obj1 is on the floor2 |
| touching | (obj1.id, obj2.id) | obj1 is touching or next to obj2
    |
| cooked | (obj1.id) | obj1 is cooked |
| burnt | (obj1.id) | obj1 is burnt |
| dusty | (obj1.id) | obj1 is dusty. If want to change from "dusty(
    obj1.id)" to "not dusty(obj1.id)", there are two ways to do it,
    depending on the available relevant objects in the scene. Either
```

```
     (1) "inside(obj1.id, dishwasher.id/sink.id)" and "toggled_on(
    dishwasher.id/sink.id)" after which the state transition will
    happen or (2) holding a cleaning tool e.g., "(holds_lh,
    scrub_brush.id/rag.id/etc.)" before taking a cleaning action
    that would induce a state change. You cannot clean without
    suitable cleaning tools. Use your common sense |
| frozen | (obj1.id) | obj1 is frozen |
| open | (obj1.id) | obj1 is open |
| sliced | (obj1.id) | obj1 is sliced. If want to change "not
    sliced(obj1.id)" to "sliced(obj1.id)", one must have a slicer. |
| soaked | (obj1.id) | obj1 is soaked |
| stained | (obj1.id) | obj1 is stained. If want to change "stained
    (obj1.id)" to "not stained(obj1.id)", there are three ways to do
     it, depending on the available relevant objects in the scene.
    Either (1) "inside(obj1.id, sink.id)" and "toggled_on(sink.id)"
    (you cannot use dishwasher for stains) after which the state
    transition will happen or (2) holding a "soaked" cleaning tool e
    .g., "soaked(scrub_brush.id/rag_id/etc. )"and "holds_lh(
    scrub_brush.id/rag.id/etc.)" before taking a cleaning action
    that would induce a state change or (3) holds detergent without
    cleaning tools e.g., "holds_rh(detergent.id)" before taking a
    cleaning action that would induce a state change |
| toggled_on | (obj1.id) | obj1 is toggled on |
| holds_rh | (obj1.id) | obj1 is in the right hand of the robot |
| holds_lh | (obj1.id) | obj1 is in the left hand of the robot |

## Available Connectives
The connectives are logical operators and quantifiers for the
    boolean expressions. They are used to combine the state
    predicates to form more comprehensive boolean expressions. Below
     is a list of available connectives and their descriptions.
| Connective Name | Arguments | Description |
| --- | --- | --- |
| and | exp1 and exp2 | evaluates to true if both exp1 and exp2 are
     true |
| or | exp1 or exp2 | evaluates to true if either exp1 or exp2 is
    true |
| not | not exp | evaluates to true if exp is false |
| forall | forall(x, exp) | evaluates to true if exp is true for
    all x |
| exists | exists(x, exp) | evaluates to true if exp is true for at
     least one x |
| forpairs | forpairs(x, y, exp) | evaluates to true if exp is true
     for all pairs of x and y. For example, forpairs(watch, basket,
    inside(watch, basket)) means that for each watch and basket, the
     watch is inside the basket. |
| forn | forn(n, x, exp) | evaluates to true if exp is true for
    exactly n times for x. For example, forn(2, jar_n_01, (not open(
    jar_n_01)) means that there are exactly two jars that are not
    open. |
| fornpairs | fornpairs(n, x, y, exp) | evaluates to true if exp is
     true for exactly n times for pairs of x and y. For example,
    fornpairs(2, watch, basket, inside(watch, basket)) means that
    there are exactly two watches inside the basket. |

# Rules You Must Follow
- Your output boolean expression list a temporal logic formula that
```

```
    describes a subgoals plan with temporal and logical order.
- The initial states are the states that are given at the beginning
    of the task.
- Your output must be a list of boolean expressions that are in
   temporal order.
- You must follow the data format and the available states and
   connectives defined above.
- The output must be consistent, logical and as detailed as
   possible. View your output as a complete state transition from
   the initial states to the final goal states.
- Please note that the robot can only hold one object in each hand.
    Also, the robot needs to have at least one hand free to perform
    any action other than put, place, take, hold. Also note that
   such actions are not part of your output but might be the
   intended state transitions between your output states. You do
   not need to describe actions in your output at all.
- Use holds_rh and holds_lh in your plan if necessary as they are
   objects' unary states. For example, stained(shoe) cannot
   directly change to not stained(shoe), but needs (partial)
   intermediate states like [soaked(rag), holds_rh(rag), not
   stained(shoe)] or [holds_rh(detergent), not stained(shoe)].
   Notice no action is needed in your output.
- Please use provided relevant objects well to help you achieve the
    final goal states. Note that inside(obj, agent) is an invalid
   state, therefore you cannot output it in your plan. You do not
   describe states of the acting agent or character.
- To make sure you output the correct results, it might help to
   reason about pseudo-actions necessary for state transitions and
   attempt planning for actions to traverse the subgoal sequence
   yourself to see of each subgoal and the final goal states are
   consistent and reachable from the initial states. You do not
   need to output this process though.

# Input content
- Relevant objects in the scene indicates the potential involvement
    those objects in the sequence action executions required to
   achieve the final goal. Each object is listed in the form{{'name
   ': <obj.id>, 'category': <object type>}}.
- Initial states are a set of state and relation predicates
   describing the initial status of each object in the scene.
- Goal states are the set of final symbolic goals we need to reach
   from the intial states.


# Input Format
## Target Task: <task identifier>
## Relevant objects in this scene
<Objects in the scene>

## Initial States
<initial_states>

## Goal States
<goal_states>

# Output format
- Your must strictly follow the json format like this {{"output":
```

```
    [<your subgoal plan>]}}, where <your subgoal plan> is a list of
    predicates presented in the temporal order.
- Each element in the list are subgoals need to be achieved at each
     temporal stage in order to reach the final goal states. The
    subgoals in the preceding temporaal stage needs to be satisfied
    before subgoals in the next temporal stage.
- For temporally interchangable subgoals, you can use connective '
    and' to combine them in the same temporal stage. If you think
    some states are equivalent, you can use connective 'or' to
    combine them in the same temporal stage.
- Do not output redundant states. A redundant state means a state
    that is either not necessary or has been satisfied before
    without taking any action.
- DO NOT INCLUDE IRRELEVANT INFORMATION (like number of line,
    explanation, etc.)

# Example:
Below we provide an example for your better understanding.
## Target Task: bottling_fruit
## Relevant objects in this scene
{{'name': 'strawberry.0', 'category': 'strawberry_n_01'}}
{{'name': 'fridge.97', 'category': 'electric_refrigerator_n_01'}}
{{'name': 'peach.0', 'category': 'peach_n_03'}}
{{'name': 'countertop.84', 'category': 'countertop_n_01'}}
{{'name': 'jar.0', 'category': 'jar_n_01'}}
{{'name': 'jar.1', 'category': 'jar_n_01'}}
{{'name': 'carving_knife.0', 'category': 'carving_knife_n_01'}}
{{'name': 'bottom_cabinet_no_top.80', 'category': 'cabinet_n_01'}}
{{'name': 'room_floor_kitchen.0', 'category': 'floor_n_01'}}

## Initial States
inside(strawberry.0, fridge.97)
inside(peach.0, fridge.97)
not sliced(strawberry.0)
not sliced(peach.0)
ontop(jar.0, countertop.84)
ontop(jar.1, countertop.84)
ontop(carving_knife.0, countertop.84)
onfloor(agent_n_01.1, room_floor_kitchen.0)

## Goal States
exists(jar_n_01, (inside(strawberry.0, jar_n_01) and (not inside(
    peach.0, jar_n_01))))
exists(jar_n_01, (inside(peach.0, jar_n_01) and (not inside(
    strawberry.0, jar_n_01))))
forall(jar_n_01, (not open(jar_n_01)))
sliced(strawberry.0)
sliced(peach.0)

## Output:
{{"output": ["ontop(strawberry.0, countertop.84) and ontop(peach.0,
     countertop.84)", "holds_rh(carving_knife.0)", "sliced(
    strawberry.0) and sliced(peach.0)", "inside(strawberry.0, jar.0)
     and not inside(peach.0, jar.0)", "inside(peach.0, jar.1) and
    not inside(strawberry.0, jar.1)", "not open(jar.0) and not open(
    jar.1)"]}}
```

```
Now, it is time for you to generate the subgoal plan for the
    following task.
# Target Task: {task_name}
## Relevant objects in this scene
{relevant_objects}

## Initial States
{initial_states}

## Goal States
{goal_states}

## Output:
```

Template 9: BEHAVIOR – Subgoal Decomposition Updated Template (2)

```
# Background Introduction
You are an expert in symbolic AI who is determining the complete
    state transitions of a household task solving by a robot. The
    objective is to list all intermediate goal states (which are
    called subgoals as a whole) to achieve the final target goals.
    The state primitives include: unary primitives describing one
    object's own unary states and binary primitives describing
    object-object relationships. The output is a list of state and
    relation predicates constructed from the primitives detailed
    below.

More specifically, your task is to read and comprehend the input,
    reason about the world model with object states and
    relationships, and relevant (pseudo-)actions for state
    transitions required to traverse the list of subgoals to achieve
     the goals in terms of symbolic predicates. The robot's planner
    will take your output and attempt to plan for action sequences
    to iteratively achieve each subgoal. Thus, your list of subgoals
     needs to follow temporal logic order so success completions of
    subgoals can lead to achieving the final goal states.

# State Primitives Vocabulary
Below we introduce the detailed data vocabulary and their format
    that you can use to generate the subgoal plan.
## Available States
The state primitive is a tuple of a predicate name and its
    arguments. Its formal definition looks like this "<PredicateName
    >(Params)", where <PredicateName> is the state name and each
    param should be ended with an id. For example, if a jar is
    opened, it is represented as "opened(jar.1)". Another example is
     "inside(coin.1, jar.1)" which describes coin.1 is inside jar.1.
     Below is a list of available states and their descriptions.
| State Name | Arguments | Description |
| --- | --- | --- |
| inside | (obj1.id, obj2.id) | obj1 is inside obj2. agent cannot
    be an argument of this state. If we have state inside(A, B), and
     you want to take A out of B you need to reach open(B) by
    openning B first |
| ontop | (obj1.id, obj2.id) | obj1 is on top of obj2 |
```

```
| nextto | (obj1.id, obj2.id) | obj1 is next to obj2 |
| under | (obj1.id, obj2.id) | obj1 is under obj2 |
| onfloor | (obj1.id, floor2.id) | obj1 is on the floor2 |
| touching | (obj1.id, obj2.id) | obj1 is touching or next to obj2
    |
| cooked | (obj1.id) | obj1 is cooked |
| burnt | (obj1.id) | obj1 is burnt |
| dusty | (obj1.id) | obj1 is dusty. If want to change from "dusty(
    obj1.id)" to "not dusty(obj1.id)", there are two ways to do it,
    depending on the available relevant objects in the scene. Either
     (1) with a series of subgoals "inside(obj1.id, dishwasher.id/
    sink.id)" followed by "toggled_on(dishwasher.id/sink.id)" after
    which the state transition will happen or (2) holding a cleaning
     tool e.g., "(holds_lh, scrub_brush.id/rag.id/etc.)" before
    taking a cleaning action that would induce a state change. You
    cannot clean without suitable cleaning tools. Use your common
    sense |
| frozen | (obj1.id) | obj1 is frozen |
| open | (obj1.id) | obj1 is open |
| sliced | (obj1.id) | obj1 is sliced. If want to change "not
    sliced(obj1.id)" to "sliced(obj1.id)", one must have a slicer. |
| soaked | (obj1.id) | obj1 is soaked |
| stained | (obj1.id) | obj1 is stained. If want to change "stained
    (obj1.id)" to "not stained(obj1.id)", there are three ways to do
     it, depending on the available relevant objects in the scene.
    Either (1) with a series of subgoals "inside(obj1.id, sink.id)"
    followed by "toggled_on(sink.id)" (you cannot use dishwasher for
     stains) after which the state transition will happen or (2)
    holding a "soaked" cleaning tool e.g., "soaked(scrub_brush.id/
    rag_id/etc. )"and "holds_lh(scrub_brush.id/rag.id/etc.)" before
    taking a cleaning action that would induce a state change or (3)
     holds detergent without cleaning tools e.g., "holds_rh(
    detergent.id)" before taking a cleaning action that would induce
     a state change |
| toggled_on | (obj1.id) | obj1 is toggled on |
| holds_rh | (obj1.id) | obj1 is in the right hand of the robot |
| holds_lh | (obj1.id) | obj1 is in the left hand of the robot |

## Available Connectives
The connectives are logical operators and quantifiers for the
    boolean expressions. They are used to combine the state
    predicates to form more comprehensive boolean expressions. Below
     is a list of available connectives and their descriptions.
    Except for "and", "or", "not", you cannot use any other
    connectives in your output.
| Connective Name | Arguments | Description |
| --- | --- | --- |
| and | exp1 and exp2 | evaluates to true if both exp1 and exp2 are
     true |
| or | exp1 or exp2 | evaluates to true if either exp1 or exp2 is
    true |
| not | not exp | evaluates to true if exp is false |
| forall | forall(x, exp) | evaluates to true if exp is true for
    all x |
| exists | exists(x, exp) | evaluates to true if exp is true for at
     least one x |
| forpairs | forpairs(x, y, exp) | evaluates to true if exp is true
```

```
     for all pairs of x and y. For example, forpairs(watch, basket,
    inside(watch, basket)) means that for each watch and basket, the
     watch is inside the basket. |
| forn | forn(n, x, exp) | evaluates to true if exp is true for
    exactly n times for x. For example, forn(2, jar_n_01, (not open(
    jar_n_01)) means that there are exactly two jars that are not
    open. |
| fornpairs | fornpairs(n, x, y, exp) | evaluates to true if exp is
     true for exactly n times for pairs of x and y. For example,
    fornpairs(2, watch, basket, inside(watch, basket)) means that
    there are exactly two watches inside the basket. |

# Rules You Must Follow
- Your output boolean expression list a temporal logic formula that
     describes a subgoals plan with temporal and logical order.
- The initial states are the states that are given at the beginning
     of the task.
- Your output must be a list of boolean expressions that are in
    temporal order.
- You must follow the data format and the available states and
    connectives defined above.
- The output must be consistent, logical and as detailed as
    possible. View your output as a complete state transition from
    the initial states to the final goal states.
- Please note that the robot can only hold one object in each hand.
     Also, the robot needs to have at least one hand free to perform
     any action other than put, place, take, hold. Also note that
    such actions are not part of your output but might be the
    intended state transitions between your output states. You do
    not need to describe actions in your output at all.
- Use holds_rh and holds_lh in your plan if necessary as they are
    objects' unary states. For example, stained(shoe) cannot
    directly change to not stained(shoe), but needs (partial)
    intermediate states like [soaked(rag), holds_rh(rag), not
    stained(shoe)] or [holds_rh(detergent), not stained(shoe)].
    Notice no action is needed in your output.
- Please use provided relevant objects well to help you achieve the
     final goal states. Note that inside(obj, agent) is an invalid
    state, therefore you cannot output it in your plan.
- Be very careful about minute itermediate subgoals such as open
    and close containers, toggle on the sink if it is off, etc.
    Those are VERY IMPORTANT subgoals.
- To make sure you output the correct results, it might help to
    reason about pseudo-actions necessary for state transitions and
    attempt planning for actions to traverse the subgoal sequence
    yourself to see of each subgoal and the final goal states are
    consistent and reachable from the initial states. You should
    expand "forall", "exists", "forpairs", "forn", "fornpairs"
    expression to multiple first-order predicates during this
    process to make sure you don't miss anything. You do not need to
     output this process though.

# Input content
- Relevant objects in the scene indicates the potential involvement
     those objects in the sequence action executions required to
    achieve the final goal. Each object is listed in the form{{'name
    ': <obj.id>, 'category': <object type>}}.
```

```
- Initial states are a set of state and relation predicates
    describing the initial status of each object in the scene.
- Goal states are the set of final symbolic goals we need to reach
    from the intial states.


# Input Format
## Target Task: <task identifier>
## Relevant objects in this scene
<Objects in the scene>

## Initial States
<initial_states>

## Goal States
<goal_states>

# Output format
- Your must strictly follow the json format like this {{"output":
    [<your subgoal plan>]}}, where <your subgoal plan> is a list of
    predicates presented in the temporal order.
- Each element in the list are subgoals need to be achieved at each
     temporal stage in order to reach the final goal states. The
    subgoals in the preceding temporaal stage needs to be satisfied
    before subgoals in the next temporal stage.
- For temporally interchangable subgoals, you can use connective '
    and' to combine them in the same temporal stage. This is
    optional and you are highly recommended to use a strict temporal
     order.
- If you think some states are equivalent, you can use connective '
    or' to combine them in the same temporal stage.
- Except for "and", "or", "not", you cannot use any other
    connectives in your output. You must instead expand "forall", "
    exists", "forpairs", "forn", "fornpairs" expressions to multiple
     first-order predicates.
- Do not output redundant states. A redundant state means a state
    that is either not necessary or has been satisfied before
    without taking any action.
- DO NOT INCLUDE IRRELEVANT INFORMATION (like number of line,
    explanation, etc.)

# Example:
Below we provide an example for your better understanding.
## Target Task: bottling_fruit
## Relevant objects in this scene
{{'name': 'strawberry.0', 'category': 'strawberry_n_01'}}
{{'name': 'fridge.97', 'category': 'electric_refrigerator_n_01'}}
{{'name': 'peach.0', 'category': 'peach_n_03'}}
{{'name': 'countertop.84', 'category': 'countertop_n_01'}}
{{'name': 'jar.0', 'category': 'jar_n_01'}}
{{'name': 'jar.1', 'category': 'jar_n_01'}}
{{'name': 'carving_knife.0', 'category': 'carving_knife_n_01'}}
{{'name': 'bottom_cabinet_no_top.80', 'category': 'cabinet_n_01'}}
{{'name': 'room_floor_kitchen.0', 'category': 'floor_n_01'}}

## Initial States
inside(strawberry.0, fridge.97)
```

```
inside(peach.0, fridge.97)
not sliced(strawberry.0)
not sliced(peach.0)
ontop(jar.0, countertop.84)
ontop(jar.1, countertop.84)
ontop(carving_knife.0, countertop.84)
onfloor(agent_n_01.1, room_floor_kitchen.0)

## Goal States
exists(jar_n_01, (inside(strawberry.0, jar_n_01) and (not inside(
    peach.0, jar_n_01))))
exists(jar_n_01, (inside(peach.0, jar_n_01) and (not inside(
    strawberry.0, jar_n_01))))
forall(jar_n_01, (not open(jar_n_01)))
sliced(strawberry.0)
sliced(peach.0)

## Output:
{{"output": ["ontop(strawberry.0, countertop.84) and ontop(peach.0,
     countertop.84)", "holds_rh(carving_knife.0)", "sliced(
    strawberry.0) and sliced(peach.0)", "inside(strawberry.0, jar.0)
     and not inside(peach.0, jar.0)", "inside(peach.0, jar.1) and
    not inside(strawberry.0, jar.1)", "not open(jar.0) and not open(
    jar.1)"]}}

Now, it is time for you to generate the subgoal plan for the
    following task.
# Target Task: {task_name}
## Relevant objects in this scene
{relevant_objects}

## Initial States
{initial_states}

## Goal States
{goal_states}

## Output:
```

Template 10: BEHAVIOR – Subgoal Decomposition Updated Template (v3)

```
'''# Background Introduction
You are an expert in symbolic AI who is determining the complete
    state transitions of a household task solving by a robot. The
    objective is to list all intermediate goal states (which are
    called subgoals as a whole) to achieve the final target goals.
    The state primitives include: unary primitives describing one
    object's own unary states and binary primitives describing
    object-object relationships. The output is a list of state and
    relation predicates constructed from the primitives detailed
    below.

More specifically, your task is to read and comprehend the input,
    reason about the world model with object states and
    relationships, and relevant (pseudo-)actions for state
```

```
    transitions required to traverse the list of subgoals to achieve
     the goals in terms of symbolic predicates. The robot's planner
    will take your output and attempt to plan for action sequences
    to iteratively achieve each subgoal. Thus, your list of subgoals
     needs to follow temporal logic order so success completions of
    subgoals can lead to achieving the final goal states.

# State Primitives Vocabulary
Below we introduce the detailed data vocabulary and their format
    that you can use to generate the subgoal plan.
## Available States
The state primitive is a tuple of a predicate name and its
    arguments. Its formal definition looks like this "<PredicateName
    >(Params)", where <PredicateName> is the state name and each
    param should be ended with an id. For example, if a jar is
    opened, it is represented as "opened(jar.1)". Another example is
     "inside(coin.1, jar.1)" which describes coin.1 is inside jar.1.
     Below is a list of available states and their descriptions.
| State Name | Arguments | Description |
| --- | --- | --- |
| inside | (obj1.id, obj2.id) | obj1 is inside obj2. agent cannot
    be an argument of this state. If we have state inside(A, B), and
     you want to take A out of B you need to reach open(B) by
    openning B first |
| ontop | (obj1.id, obj2.id) | obj1 is on top of obj2 |
| nextto | (obj1.id, obj2.id) | obj1 is next to obj2 |
| under | (obj1.id, obj2.id) | obj1 is under obj2 |
| onfloor | (obj1.id, floor2.id) | obj1 is on the floor2 |
| touching | (obj1.id, obj2.id) | obj1 is touching or next to obj2
    |
| cooked | (obj1.id) | obj1 is cooked |
| burnt | (obj1.id) | obj1 is burnt |
| dusty | (obj1.id) | obj1 is dusty. If want to change from "dusty(
    obj1.id)" to "not dusty(obj1.id)", there are two ways to do it,
    depending on the available relevant objects in the scene. Either
     (1) with a series of subgoals "toggled_on(dishwasher.id/sink.id
    )" followed by "inside(obj1.id, dishwasher.id/sink.id)" after
    which the state transition will happen or (2) holding a cleaning
     tool e.g., "(holds_lh, scrub_brush.id/rag.id/etc.)" before
    taking a cleaning action that would induce a state change. You
    cannot clean without suitable cleaning tools. Use your common
    sense |
| frozen | (obj1.id) | obj1 is frozen |
| open | (obj1.id) | obj1 is open. This state is a preconditions
    for many "inside" states. Objects with out a lid such as sink,
    teapot, bucket, trashcan, shelf, saucepan, bathtub, casserole,
    dish, bowl, carton do not need to be opened and cannot use this
    primitive. |
| sliced | (obj1.id) | obj1 is sliced. If want to change "not
    sliced(obj1.id)" to "sliced(obj1.id)", one must have a slicer. |
| soaked | (obj1.id) | obj1 is soaked. For an object to be soaked,
    you first need to always make sure you satisfy "toggled_on(sink.
    id)" then "inside(obj1.id, sink.id)". Only after that the object
     can be soaked. You can also soak teabags in a boiling teapot.
    So you only need to satisfy the series "ontop(teapot.id, stove.
    id)" -> "toggled_on(stove.id)" -> "inside(obj1.id, teapot.id)" |
| stained | (obj1.id) | obj1 is stained. If want to change "stained
```

```
    (obj1.id)" to "not stained(obj1.id)", there are three ways to do
     it, depending on the available relevant objects in the scene.
    Either (1) with a series of subgoals "toggled_on(sink.id)"
    followed by "inside(obj1.id, sink.id)" (you cannot use
    dishwasher for stains) after which the state transition will
    happen or (2) holding a "soaked" cleaning tool e.g., "soaked(
    scrub_brush.id/rag_id/etc. )"and "holds_lh(scrub_brush.id/rag.id
    /etc.)" before taking a cleaning action that would induce a
    state change or (3) holds detergent without cleaning tools by
    satisfying "holds_rh(detergent.id)" before taking a cleaning
    action that would induce a state change |
| toggled_on | (obj1.id) | obj1 is toggled on |
| holds_rh | (obj1.id) | obj1 is in the right hand of the robot |
| holds_lh | (obj1.id) | obj1 is in the left hand of the robot |

## Available Connectives
The connectives are logical operators and quantifiers for the
    boolean expressions. They are used to combine the state
    predicates to form more comprehensive boolean expressions. Below
     is a list of available connectives and their descriptions.
    Except for "and", "or", "not", you cannot use any other
    connectives in your output.
| Connective Name | Arguments | Description |
| --- | --- | --- |
| and | exp1 and exp2 | evaluates to true if both exp1 and exp2 are
     true |
| or | exp1 or exp2 | evaluates to true if either exp1 or exp2 is
    true |
| not | not exp | evaluates to true if exp is false |
| forall | forall(x, exp) | evaluates to true if exp is true for
    all x |
| exists | exists(x, exp) | evaluates to true if exp is true for at
     least one x |
| forpairs | forpairs(x, y, exp) | evaluates to true if exp is true
     for all pairs of x and y. For example, forpairs(watch, basket,
    inside(watch, basket)) means that for each watch and basket, the
     watch is inside the basket. |
| forn | forn(n, x, exp) | evaluates to true if exp is true for
    exactly n times for x. For example, forn(2, jar_n_01, (not open(
    jar_n_01)) means that there are exactly two jars that are not
    open. |
| fornpairs | fornpairs(n, x, y, exp) | evaluates to true if exp is
     true for exactly n times for pairs of x and y. For example,
    fornpairs(2, watch, basket, inside(watch, basket)) means that
    there are exactly two watches inside the basket. |

# Rules You Must Follow
- Your output boolean expression list a temporal logic formula that
     describes a subgoals plan with temporal and logical order.
- The initial states are the states that are given at the beginning
     of the task.
- Your output must be a list of boolean expressions that are in
    temporal order.
- You must follow the data format and the available states and
    connectives defined above.
- The output must be consistent, logical and as detailed as
    possible. View your output as a complete state transition from
```

```
    the initial states to the final goal states.
- Please note that the robot can only hold one object in each hand.
     Also, the robot needs to have at least one hand free to perform
     any action other than put, place, take, hold. Also note that
    such actions are not part of your output but might be the
    intended state transitions between your output states. You do
    not need to describe actions in your output at all.
- Use holds_rh and holds_lh in your plan if necessary as they are
    objects' unary states. For example, stained(shoe) cannot
    directly change to not stained(shoe), but needs (partial)
    intermediate states like [soaked(rag), holds_rh(rag), not
    stained(shoe)] or [holds_rh(detergent), not stained(shoe)].
    Notice no action is needed in your output.
- Please use provided relevant objects well to help you achieve the
     final goal states. Note that inside(obj, agent) is an invalid
    state, therefore you cannot output it in your plan.
- Be very careful about minute itermediate subgoals such as open
    and close containers, toggle on the sink if it is off, etc.
    Those are VERY IMPORTANT subgoals.
- To make sure you output the correct results, reason about pseudo-
    actions necessary for state transitions and attempt planning for
     actions to traverse the subgoal sequence yourself to see of
    each subgoal and the final goal states are consistent and
    reachable from the initial states. You should expand "forall", "
    exists", "forpairs", "forn", "fornpairs" expression to multiple
    first-order predicates during this process to make sure you don'
    t miss anything. You do not need to output this process though.

# Input content
- The name of the target task can give hints on the priority of
    subgoals to be achieved in temporal order.
- Relevant objects in the scene indicates the potential involvement
     those objects in the sequence action executions required to
    achieve the final goal. Each object is listed in the form{{'name
    ': <obj.id>, 'category': <object type>}}.
- Initial states are a set of state and relation predicates
    describing the initial status of each object in the scene.
- Goal states are the set of final symbolic goals we need to reach
    from the intial states.


# Input Format
## Target Task: <task identifier>
## Relevant objects in this scene
<Objects in the scene>

## Initial States
<initial_states>

## Goal States
<goal_states>

# Output format
- Your must strictly follow the json format like this {{"output":
    [<your subgoal plan>]}}, where <your subgoal plan> is a list of
    predicates presented in the temporal order.
- Each element in the list are subgoals need to be achieved at each
```

```
     temporal stage in order to reach the final goal states. The
    subgoals in the preceding temporaal stage needs to be satisfied
    before subgoals in the next temporal stage.
- For temporally interchangable subgoals, you can use connective '
    and' to combine them in the same temporal stage. This is
    optional and you are highly recommended to use a strict temporal
     order.
- If you think some states are equivalent, you can use connective '
    or' to combine them in the same temporal stage.
- Except for "and", "or", "not", you cannot use any other
    connectives in your output. You must instead expand "forall", "
    exists", "forpairs", "forn", "fornpairs" expressions to multiple
     first-order predicates.
- The final goal states must be parts of your subgoal plan.
- Do not output redundant states. A redundant state means a state
    that is either not necessary or has been satisfied before
    without taking any action.
- DO NOT INCLUDE IRRELEVANT INFORMATION (like number of line,
    explanation, etc.)

# Example:
Below we provide an example for your better understanding.
## Target Task: bottling_fruit
## Relevant objects in this scene
{{'name': 'strawberry.0', 'category': 'strawberry_n_01'}}
{{'name': 'fridge.97', 'category': 'electric_refrigerator_n_01'}}
{{'name': 'peach.0', 'category': 'peach_n_03'}}
{{'name': 'countertop.84', 'category': 'countertop_n_01'}}
{{'name': 'jar.0', 'category': 'jar_n_01'}}
{{'name': 'jar.1', 'category': 'jar_n_01'}}
{{'name': 'carving_knife.0', 'category': 'carving_knife_n_01'}}
{{'name': 'bottom_cabinet_no_top.80', 'category': 'cabinet_n_01'}}
{{'name': 'room_floor_kitchen.0', 'category': 'floor_n_01'}}

## Initial States
inside(strawberry.0, fridge.97)
inside(peach.0, fridge.97)
not sliced(strawberry.0)
not sliced(peach.0)
ontop(jar.0, countertop.84)
ontop(jar.1, countertop.84)
ontop(carving_knife.0, countertop.84)
onfloor(agent_n_01.1, room_floor_kitchen.0)

## Goal States
exists(jar_n_01, (inside(strawberry.0, jar_n_01) and (not inside(
    peach.0, jar_n_01))))
exists(jar_n_01, (inside(peach.0, jar_n_01) and (not inside(
    strawberry.0, jar_n_01))))
forall(jar_n_01, (not open(jar_n_01)))
sliced(strawberry.0)
sliced(peach.0)

## Output:
{{"output": ["ontop(strawberry.0, countertop.84) and ontop(peach.0,
     countertop.84)", "holds_rh(carving_knife.0)", "sliced(
    strawberry.0) and sliced(peach.0)", "inside(strawberry.0, jar.0)
```

```
     and not inside(peach.0, jar.0)", "inside(peach.0, jar.1) and
    not inside(strawberry.0, jar.1)", "not open(jar.0) and not open(
    jar.1)"]}}

Now, it is time for you to generate the subgoal plan for the
    following task.
# Target Task: {task_name}
## Relevant objects in this scene
{relevant_objects}

## Initial States
{initial_states}

## Goal States
{goal_states}

## Output:
```

Template 11: BEHAVIOR – Subgoal Decomposition Updated Template (v4)

```
# Background Introduction
You are an expert in symbolic AI and PDDL expert who is determining
     the complete state transitions of a household task to be solved
     by a robot. The objective is to derive an action plan to
    achieve the goal states from the initial states and list all
    intermediate states yields by each action (the effects of each
    action) in the plan to achieve the final target goals. These
    intermediate states are called subgoals. The state primitives
    include: unary primitives describing one object's own unary
    states and binary primitives describing object-object
    relationships. The output is a list of state and relation
    predicates constructed from the primitives detailed below.

More specifically, your task is to read and comprehend the input,
    reason about the world model with object states and
    relationships, and relevant (pseudo-)actions for the goals. Then
     derive a sequence of actions needed to achieve the goals, but
    only provide the list of effects of those actions as a list of
    subgoals to achieve the goals in terms of symbolic predicates.
    The robot's planner will take your output and attempt to plan
    for action sequences to iteratively achieve each subgoal. Thus,
    your list of subgoals needs to follow temporal logic order so
    success completions of subgoals can lead to achieving the final
    goal states.

# State Primitives Vocabulary
Below we introduce the detailed data vocabulary and their format
    that you can use to generate the subgoal plan.
## Available States
The state primitive is a tuple of a predicate name and its
    arguments. Its formal definition looks like this "<PredicateName
    >(Params)", where <PredicateName> is the state name and each
    param should be ended with an id. For example, if a jar is
    opened, it is represented as "opened(jar.1)". Another example is
     "inside(coin.1, jar.1)" which describes coin.1 is inside jar.1.
```

```
    Below is a list of available states and their descriptions.
| State Name | Arguments | Description |
| --- | --- | --- |
| inside | (obj1.id, obj2.id) | obj1 is inside obj2. agent cannot
   be an argument of this state. If we have state inside(A, B), and
    you want to take A out of B you need to reach open(B) by
   openning B first |
| ontop | (obj1.id, obj2.id) | obj1 is on top of obj2 |
| nextto | (obj1.id, obj2.id) | obj1 is next to obj2 |
| under | (obj1.id, obj2.id) | obj1 is under obj2 |
| onfloor | (obj1.id, floor2.id) | obj1 is on the floor2 |
| touching | (obj1.id, obj2.id) | obj1 is touching or next to obj2
   |
| cooked | (obj1.id) | obj1 is cooked |
| burnt | (obj1.id) | obj1 is burnt |
| dusty | (obj1.id) | obj1 is dusty. If want to change from "dusty(
   obj1.id)" to "not dusty(obj1.id)", there are two ways to do it,
   depending on the available relevant objects in the scene. Either
    (1) with a series of subgoals "toggled_on(dishwasher.id/sink.id
   )" followed by "inside(obj1.id, dishwasher.id/sink.id)" after
   which the state transition will happen or (2) holding a cleaning
    tool e.g., "(holds_lh, scrub_brush.id/rag.id/etc.)" before
   taking a cleaning action that would induce a state change. You
   cannot clean without suitable cleaning tools. Use your common
   sense |
| frozen | (obj1.id) | obj1 is frozen |
| open | (obj1.id) | obj1 is open. This state is a preconditions
   for many "inside" states. Objects with out a lid such as sink,
   teapot, bucket, trashcan, shelf, saucepan, bathtub, casserole,
   dish, bowl, carton do not need to be opened and cannot use this
   primitive. |
| sliced | (obj1.id) | obj1 is sliced. If want to change "not
   sliced(obj1.id)" to "sliced(obj1.id)", one must have a slicer. |
| soaked | (obj1.id) | obj1 is soaked. For an object to be soaked,
   you first need to always make sure you satisfy "toggled_on(sink.
   id)" then "inside(obj1.id, sink.id)". Only after that the object
    can be soaked. You can also soak teabags in a boiling teapot.
   So you only need to satisfy the series "ontop(teapot.id, stove.
   id)" -> "toggled_on(stove.id)" -> "inside(obj1.id, teapot.id)" |
| stained | (obj1.id) | obj1 is stained. If want to change "stained
   (obj1.id)" to "not stained(obj1.id)", there are three ways to do
    it, depending on the available relevant objects in the scene.
   Either (1) with a series of subgoals "toggled_on(sink.id)"
   followed by "inside(obj1.id, sink.id)" (you cannot use
   dishwasher for stains) after which the state transition will
   happen or (2) holding a "soaked" cleaning tool e.g., "soaked(
   scrub_brush.id/rag_id/etc. )"and "holds_lh(scrub_brush.id/rag.id
   /etc.)" before taking a cleaning action that would induce a
   state change or (3) holds detergent without cleaning tools by
   satisfying "holds_rh(detergent.id)" before taking a cleaning
   action that would induce a state change |
| toggled_on | (obj1.id) | obj1 is toggled on |
| holds_rh | (obj1.id) | obj1 is in the right hand of the robot |
| holds_lh | (obj1.id) | obj1 is in the left hand of the robot |

## Available Connectives
The connectives are logical operators and quantifiers for the
```

```
    boolean expressions. They are used to combine the state
    predicates to form more comprehensive boolean expressions. Below
     is a list of available connectives and their descriptions.
    Except for "and", "or", "not", you cannot use any other
    connectives in your output.
| Connective Name | Arguments | Description |
| --- | --- | --- |
| and | exp1 and exp2 | evaluates to true if both exp1 and exp2 are
     true |
| or | exp1 or exp2 | evaluates to true if either exp1 or exp2 is
    true |
| not | not exp | evaluates to true if exp is false |
| forall | forall(x, exp) | evaluates to true if exp is true for
    all x |
| exists | exists(x, exp) | evaluates to true if exp is true for at
     least one x |
| forpairs | forpairs(x, y, exp) | evaluates to true if exp is true
     for all pairs of x and y. For example, forpairs(watch, basket,
    inside(watch, basket)) means that for each watch and basket, the
     watch is inside the basket. |
| forn | forn(n, x, exp) | evaluates to true if exp is true for
    exactly n times for x. For example, forn(2, jar_n_01, (not open(
    jar_n_01)) means that there are exactly two jars that are not
    open. |
| fornpairs | fornpairs(n, x, y, exp) | evaluates to true if exp is
     true for exactly n times for pairs of x and y. For example,
    fornpairs(2, watch, basket, inside(watch, basket)) means that
    there are exactly two watches inside the basket. |

# Rules You Must Follow
- Your output boolean expression list a temporal logic formula that
     describes a subgoals plan with temporal and logical order.
- The initial states are the states that are given at the beginning
     of the task.
- Your output must be a list of boolean expressions that are in
    temporal order.
- You must follow the data format and the available states and
    connectives defined above.
- The output must be consistent, logical and as detailed as
    possible. View your output as a complete state transition from
    the initial states to the final goal states.
- Please note that the robot can only hold one object in each hand.
     Also, the robot needs to have at least one hand free to perform
     any action other than put, place, take, hold. Also note that
    such actions are not part of your output but might be the
    intended state transitions between your output states. You do
    not need to describe actions in your output at all.
- Use holds_rh and holds_lh in your plan if necessary as they are
    objects' unary states. For example, stained(shoe) cannot
    directly change to not stained(shoe), but needs (partial)
    intermediate states like [soaked(rag), holds_rh(rag), not
    stained(shoe)] or [holds_rh(detergent), not stained(shoe)].
    Notice no action is needed in your output.
- Please use provided relevant objects well to help you achieve the
     final goal states. Note that inside(obj, agent) is an invalid
    state, therefore you cannot output it in your plan.
- Be very careful about minute itermediate subgoals such as open
```

```
    and close containers, toggle on the sink if it is off, etc.
    Those are VERY IMPORTANT subgoals.
- To make sure you output the correct results, reason about pseudo-
    actions necessary for state transitions and attempt planning for
     actions to traverse the subgoal sequence yourself to see of
    each subgoal and the final goal states are consistent and
    reachable from the initial states. You should expand "forall", "
    exists", "forpairs", "forn", "fornpairs" expression to multiple
    first-order predicates during this process to make sure you don'
    t miss anything. You do not need to output this process though.

# Input content
- The name of the target task can give hints on the priority of
    subgoals to be achieved in temporal order.
- Relevant objects in the scene indicates the potential involvement
     those objects in the sequence action executions required to
    achieve the final goal. Each object is listed in the form{{'name
    ': <obj.id>, 'category': <object type>}}.
- Initial states are a set of state and relation predicates
    describing the initial status of each object in the scene.
- Goal states are the set of final symbolic goals we need to reach
    from the intial states.


# Input Format
## Target Task: <task identifier>
## Relevant objects in this scene
<Objects in the scene>

## Initial States
<initial_states>

## Goal States
<goal_states>

# Output format
- Your must strictly follow the json format like this {{"output":
    [<your subgoal plan>]}}, where <your subgoal plan> is a list of
    predicates presented in the temporal order.
- Each element in the list are subgoals need to be achieved at each
     temporal stage in order to reach the final goal states. The
    subgoals in the preceding temporaal stage needs to be satisfied
    before subgoals in the next temporal stage.
- For temporally interchangable subgoals, you can use connective '
    and' to combine them in the same temporal stage. This is
    optional and you are highly recommended to use a strict temporal
     order.
- If you think some states are equivalent, you can use connective '
    or' to combine them in the same temporal stage.
- Except for "and", "or", "not", you cannot use any other
    connectives in your output. You must instead expand "forall", "
    exists", "forpairs", "forn", "fornpairs" expressions to multiple
     first-order predicates.
- The final goal states must be parts of your subgoal plan.
- Do not output redundant states. A redundant state means a state
    that is either not necessary or has been satisfied before
    without taking any action.
```

```
- DO NOT INCLUDE IRRELEVANT INFORMATION (like number of line,
    explanation, etc.)

# Example:
Below we provide an example for your better understanding.
## Target Task: bottling_fruit
## Relevant objects in this scene
{{'name': 'strawberry.0', 'category': 'strawberry_n_01'}}
{{'name': 'fridge.97', 'category': 'electric_refrigerator_n_01'}}
{{'name': 'peach.0', 'category': 'peach_n_03'}}
{{'name': 'countertop.84', 'category': 'countertop_n_01'}}
{{'name': 'jar.0', 'category': 'jar_n_01'}}
{{'name': 'jar.1', 'category': 'jar_n_01'}}
{{'name': 'carving_knife.0', 'category': 'carving_knife_n_01'}}
{{'name': 'bottom_cabinet_no_top.80', 'category': 'cabinet_n_01'}}
{{'name': 'room_floor_kitchen.0', 'category': 'floor_n_01'}}

## Initial States
inside(strawberry.0, fridge.97)
inside(peach.0, fridge.97)
not sliced(strawberry.0)
not sliced(peach.0)
ontop(jar.0, countertop.84)
ontop(jar.1, countertop.84)
ontop(carving_knife.0, countertop.84)
onfloor(agent_n_01.1, room_floor_kitchen.0)

## Goal States
exists(jar_n_01, (inside(strawberry.0, jar_n_01) and (not inside(
    peach.0, jar_n_01))))
exists(jar_n_01, (inside(peach.0, jar_n_01) and (not inside(
    strawberry.0, jar_n_01))))
forall(jar_n_01, (not open(jar_n_01)))
sliced(strawberry.0)
sliced(peach.0)

## Output:
{{"output": ["holds_lh(strawberry.0)", "holds_rh(peach.0)", "ontop(
    strawberry.0, countertop.84) and (not holds_lh(strawberry.0))",
    "ontop(peach.0, counter.84) and (not holds_rh(peach.0))",
    holds_rh(carving_knife.0)", "sliced(strawberry.0) and sliced(
    peach.0)", "open(jar.0)", "holds_lh(strawberry.0)", "inside(
    strawberry.0, jar.0) and not inside(peach.0, jar.0) and not
    holds_lh(strawberry.0)", "not open(jar.0)", "open(jar.1)", "
    holds_lh(peach.0)", "inside(peach.0, jar.1) and not inside(
    strawberry.0, jar.1) and not holds_lh(peach.0)", "not open(jar
    .1)"]}}

## Explanation (not part of the output):
The action plan for the example is
left_grasp(strawberry.0) -> effect: holds_lh(strawberry.0)
right_grasp(peach.0) -> effect: holds_rh(peach.0)
left_place_ontop(strawberry.0, counter.84) -> effect: ontop(
    strawberry.0, counter.84) and (not holds_lh(strawberry.0))
right_place_ontop(peach.0, counter.84) -> effect: ontop(peach.0,
    counter.84) and (not holds_rh(peach.0))
right_grasp(carving_knife.0) -> effect: holds_rh(carving_knife.0)
```

```
slice(strawberry.0) -> effect: sliced(strawberry.0)
slice(peach.0) -> effect: sliced(peach.0)
Since the jars are already on the countertop which is in reach of
    the agent, and we have one free left hand, we can just open the
    jar.
open(jar.0) -> effect: open(jar.0)
Since the agent have not released the knife in the right hand, he
    can only use his left hand.
left_grasp(strawberry.0) -> effect: holds_lh(strawberry.0)
left_place_inside(strawberry.0, jar.0) -> effect: inside(strawberry
    .0, jar.0) and (not holds_lh(strawberry.0))
close(jar.0) -> effect: not open(jar.0)
open(jar.1) -> effect: open(jar.1)
left_grasp(peach.0) -> effect: holds_lh(peach.0)
left_place_inside(peach.0, jar.1) -> effect: inside(peach.0, jar.1)
     and (not holds_lh(peach.0))
close(jar.1) -> effect: not open(jar.1)

Now, it is time for you to generate the subgoal plan for the
    following task. Strictly follow the output format. No
    explanation needed.
# Target Task: {task_name}
## Relevant objects in this scene
{relevant_objects}

## Initial States
{initial_states}

## Goal States
{goal_states}

## Output:
```

D.2.2 VirtualHome

Template 12: VirtualHome – Subgoal Decomposition Original Template

```
# Background Introduction
You are determining complete state transitions of a household task
    solving by a robot. The goal is to list all intermediate states
    and necessary actions in temporal order to achieve the target
    goals. The output consists of Boolean expressions, which are
    comprised of state and action primitives. Here, a state or
    action primitive is a first-order predicate as combinition of a
    predicate name and its parameters. Please note that do not use
    actions in your output unless necessary.In short, your task is
    to output the subgoal plan in the required format.

# Data Vocabulary Introduction
## Available States
State primitive is a tuple of a predicate name and its arguments.
    Its formal definition looks like this "<PredicateName>(Params)",
     where <PredicateName> is the state name and each param should
    be ended with an id. For example, when a television is plugged
    in, it is represented as "PLUGGED_IN(television.1). Another
```

```
    example is, if character is facing a television, it is
    represented as "FACING(character.1, television.1)". Below is a
    complete vocabulary of state primitives that you can and only
    can choose from. Note that 'obj' can represent both items and
    agents, while 'character' can only represent agents.
| Predicate Name | Arguments | Description |
| --- | --- | --- |
| CLOSED | (obj1.id) | obj1 is closed |
| OPEN | (obj1.id) | obj1 is open |
| ON | (obj1.id) | obj1 is turned on, or it is activated |
| OFF | (obj1.id) | obj1 is turned off, or it is deactivated |
| PLUGGED_IN | (obj1.id) | obj1 is plugged in |
| PLUGGED_OUT | (obj1.id) | obj1 is unplugged |
| SITTING | (character1.id) | character1 is sitting, and this
    represents a state of a character |
| LYING | (character1.id) | character1 is lying |
| CLEAN | (obj1.id) | obj1 is clean |
| DIRTY | (obj1.id) | obj1 is dirty |
| ONTOP | (obj1.id, obj2.id) | obj1 is on top of obj2 |
| INSIDE | (obj1.id, obj2.id) | obj1 is inside obj2 |
| BETWEEN | (obj1.id, obj2.id, obj3.id) | obj1 is between obj2 and
    obj3 |
| NEXT_TO | (obj1.id, obj2.id) | obj1 is close to or next to obj2 |
| FACING | (character1.id, obj1.id) | character1 is facing obj1 |
| HOLDS_RH | (character1.id, obj1.id) | character1 is holding obj1
    with right hand |
| HOLDS_LH | (character1.id, obj1.id) | character1 is holding obj1
    with left hand |
## Available Actions
Action primitive is similar to state primitive. Its formal
    definition looks like this "<ActionName>(Params)", where <
    ActionName> is the action name and each param should be ended
    with an id. Note that, you do not need to list actions in most
    cases. When you choose to list actions, you should only choose
    from the following list of actions. For other cases, use state
    predicate as substitutes. Here, 'obj' only refers to items, not
    agents.
| Action Name | Arguments | Argument Restriction | Description |
| --- | --- | --- | --- |
| DRINK | (obj1.id) | obj1 is ['DRINKABLE', 'RECIPIENT'] | drinks
    obj1, need to hold obj1 first |
| EAT | (obj1.id) | obj1 is ['EATABLE'] | eats obj1, need to hold
    obj1 first |
| CUT | (obj1.id) | obj1 is ['EATABLE', 'CUTABLE'] | cuts obj1,
    obj1 is food|
| TOUCH | (obj1.id) | none | touches obj1 |
| LOOKAT | (obj1.id) | none | looks at obj1, it has a precondition
    that agent should be facing at obj1 first |
| WATCH | (obj1.id) | none | watches obj1 |
| READ | (obj1.id) | obj1 is ['READABLE'] | reads obj1, need to
    hold obj1 first |
| TYPE | (obj1.id) | obj1 is ['HAS_SWITCH'] | types on obj1 |
| PUSH | (obj1.id) | obj1 is ['MOVABLE'] | pushes obj1 |
| PULL | (obj1.id) | obj1 is ['MOVABLE'] | pulls obj1 |
| MOVE | (obj1.id) | obj1 is ['MOVABLE'] | moves obj1 |
| SQUEEZE | (obj1.id) | obj1 is ['CLOTHES'] | squeezes obj1 |
| SLEEP | none | none | sleeps, need to be at LYING or SITTING
```

```
    first |
| WAKEUP | none | none | wakes up, need to be at LYING or SITTING
    first |
| RINSE | (obj1.id) | none | rinses obj1, use only for cleaning
    appliances or teeth |
| SCRUB | (obj1.id) | none | scrubs obj1, use only for cleaning
    appliances or teeth |
| WASH | (obj1.id) | none | washes obj1, use only for appliances |
| GRAB | (obj1.id) | obj1 is ['GRABBABLE'] | grabs obj1 |
| SWITCHOFF | (obj1.id) | obj1 is ['HAS_SWITCH'] | switches off
    obj1 |
| POUR | (obj1.id, obj2.id) | none | pours obj1 into obj2 |

# Rules You Must Follow
- Temporal logic formula refers to a Boolean expression that
    describes a subgoals plan with temporal and logical order.
- The atomic Boolean expression includes both state primitive and
    action primitive.
- Boolean expressions in the same line are interchangeable with no
    temporal order requirement.
- Boolean expresssions in different lines are in temporal order,
    where the first one should be satisfied before the second one.
- Boolean expression can be combined with the following logical
    operators: "and", "or".
- The "and" operator combines Boolean expressions that are
    interchangeable but needs to be satisfied simultaneously in the
    end.
- The "or" operator combines Boolean expressions that are
    interchangeable but only one of them needs to be satisfied in
    the end.
- When there is temporal order requirement, output the Boolean
    expressions in different lines.
- Add intermediate states if necessary to improve logical
    consistency.
- If you want to change state of A, while A is in B and B is closed,
     you should make sure B is open first.
- Your output format should strictly follow this json format: {{"
    necessity_to_use_action": <necessity>, "actions_to_include": [<
    actions>], "output": [<your subgoal plan>]}}, where in <
    necessity> you should put "yes" or "no" to indicate whether
    actions should be included in subgoal plans. If you believe it
    is necessary to use actions, in the field <actions>, you should
    list all actions you used in your output. Otherwise, you should
    simply output an empty list []. In the field <your subgoal plan
    >, you should list all Boolean expressions in the required
    format and the temporal order.

Below are two examples for your better understanding.
## Example 1: Task category is "Listen to music"
## Relevant Objects in the Scene
| obj | category | properties |
| --- | --- | --- |
| bathroom.1 | Rooms | [] |
| character.65 | Characters | [] |
| home_office.319 | Rooms | [] |
| couch.352 | Furniture | ['LIEABLE', 'MOVABLE', 'SITTABLE', '
    SURFACES'] |
```

```
| television.410 | Electronics | ['HAS_PLUG', 'HAS_SWITCH', '
    LOOKABLE'] |
| dvd_player.1000 | placable_objects | ['CAN_OPEN', 'GRABBABLE', '
    HAS_PLUG', 'HAS_SWITCH', 'MOVABLE', 'SURFACES'] |

## Initial States
CLEAN(dvd_player.1000)
CLOSED(dvd_player.1000)
OFF(dvd_player.1000)
PLUGGED_IN(dvd_player.1000)
INSIDE(character.65, bathroom.1)

## Goal States
[States]
CLOSED(dvd_player.1000)
ON(dvd_player.1000)
PLUGGED_IN(dvd_player.1000)
[Actions Must Include]: Actions are listed in the execution order,
    each line is one action to satisfy. If "A or B or ..." is
    presented in one line, then only one of them needs to be
    satisfied.
None

## Necessity to Use Actions
No

## Output: Based on initial states in this task, achieve final goal
     states logically and reasonably. It does not matter which state
     should be satisfied first, as long as all goal states can be
    satisfied at the end. Make sure your output follows the json
    format, and do not include irrelevant information, do not
    include any explanation.
{{"necessity_to_use_action": "no", "actions_to_include": [], "
    output": ["NEXT_TO(character.65, dvd_player.1000)", "FACING(
    character.65, dvd_player.1000)", "PLUGGED_IN(dvd_player.1000)
    and CLOSED(dvd_player.1000)", "ON(dvd_player.1000)"]}}

# Example 2: Task category is "Browse internet"
## Relevant Objects in the Scene
| bathroom.1 | Rooms | [] |
| character.65 | Characters | [] |
| floor.208 | Floor | ['SURFACES'] |
| wall.213 | Walls | [] |
| home_office.319 | Rooms | [] |
| floor.325 | Floors | ['SURFACES'] |
| floor.326 | Floors | ['SURFACES'] |
| wall.330 | Walls | [] |
| wall.331 | Walls | [] |
| doorjamb.346 | Doors | [] |
| walllamp.351 | Lamps | [] |
| chair.356 | Furniture | ['GRABBABLE', 'MOVABLE', 'SITTABLE', '
    SURFACES'] |
| desk.357 | Furniture | ['MOVABLE', 'SURFACES'] |
| powersocket.412 | Electronics | [] |
| mouse.413 | Electronics | ['GRABBABLE', 'HAS_PLUG', 'MOVABLE'] |
| mousepad.414 | Electronics | ['MOVABLE', 'SURFACES'] |
| keyboard.415 | Electronics | ['GRABBABLE', 'HAS_PLUG', 'MOVABLE']
```

```
     |
| cpuscreen.416 | Electronics | [] |
| computer.417 | Electronics | ['HAS_SWITCH', 'LOOKABLE'] |

## Initial States
CLEAN(computer.417)
OFF(computer.417)
ONTOP(mouse.413, mousepad.414)
ONTOP(mouse.413, desk.357)
ONTOP(keyboard.415, desk.357)
INSIDE(character.65, bathroom.1)

## Goal States
[States]
ON(computer.417)
INSIDE(character.65, home_office.319)
HOLDS_LH(character.65, keyboard.415)
FACING(character.65, computer.417)
HOLDS_RH(character.65, mouse.413)
[Actions Must Include]: Actions are listed in the execution order,
    each line is one action to satisfy. If "A or B or ..." is
    presented in one line, then only one of them needs to be
    satisfied.
LOOKAT or WATCH

## Necessity to Use Actions
Yes

## Output: Based on initial states in this task, achieve final goal
     states logically and reasonably. It does not matter which state
     should be satisfied first, as long as all goal states can be
    satisfied at the end. Make sure your output follows the json
    format. Do not include irrelevant information, only output json
    object.
{{"necessity_to_use_action": "yes", "actions_to_include": ["LOOKAT
    "], "output": ["NEXT_TO(character.65, computer.417)", "ONTOP(
    character.65, chair.356)", "HOLDS_RH(character.65, mouse.413)
    and HOLDS_LH(character.65, keyboard.415)", "FACING(character.65,
     computer.417)", "LOOKAT(computer.417)"]}}

Now, it is time for you to generate the subgoal plan for the
    following task.
# Target Task: Task category is {task_name}
## Relevant Objects in the Scene
{relevant_objects}

## Initial States
{initial_states}

## Goal States
[States]
{final_states}
[Actions Must Include]: Actions are listed in the execution order,
    each line is one action to satisfy. If "A or B or ..." is
    presented in one line, then only one of them needs to be
    satisfied.
{final_actions}
```

```
## Necessity to Use Actions
{necessity}

## Output: Based on initial states in this task, achieve final goal
     states logically and reasonably. It does not matter which state
     should be satisfied first, as long as all goal states can be
    satisfied at the end. Make sure your output follows the json
    format. Do not include irrelevant information, only output json
    object.
```

Template 13: VirtualHome – Subgoal Decomposition Updated Template (v1)

```
# Background Introduction
You are determining complete state transitions of a household task
    solving by a robot. The objective is to list all intermediate
    goal states in temporal order to achieve the target goals. The
    state primitives include: unary primitives describing one object
    's own unary states, binary primitives describing object-object
    relationships, and action primitives describing ongoing actions
    during the state. The output consists of Boolean expressions,
    which are comprised of state and action primitives. Here, a
    state or action primitive is a first-order predicate as
    combination of a predicate name and its parameters. Please note
    that do not use actions in your output if you can fully describe
     goal states with allowable state primitives only. In short,
    your task is to output the subgoal plan in the required format.

# Data Vocabulary Introduction
## Available States
State primitive is a tuple of a predicate name and its arguments.
    Its formal definition looks like this "<PredicateName>(Params)",
     where <PredicateName> is the state name and each param should
    be ended with an id. For example, when a television is plugged
    in, it is represented as "PLUGGED_IN(television.1). Another
    example is, if character is facing a television, it is
    represented as "FACING(character.1, television.1)". Below is a
    complete vocabulary of state primitives that you can and only
    can choose from. Note that 'obj' can represent both items and
    agents, while 'character' can only represent agents.
| Predicate Name | Arguments | Description |
| --- | --- | --- |
| CLOSED | (obj1.id) | obj1 is closed |
| OPEN | (obj1.id) | obj1 is open |
| ON | (obj1.id) | obj1 is turned on, or it is activated |
| OFF | (obj1.id) | obj1 is turned off, or it is deactivated |
| PLUGGED_IN | (obj1.id) | obj1 is plugged in |
| PLUGGED_OUT | (obj1.id) | obj1 is unplugged |
| SITTING | (character1.id) | character1 is sitting, and this
    represents a state of a character |
| LYING | (character1.id) | character1 is lying |
| CLEAN | (obj1.id) | obj1 is clean |
| DIRTY | (obj1.id) | obj1 is dirty |
| ONTOP | (obj1.id, obj2.id) | obj1 is on top of obj2 |
| INSIDE | (obj1.id, obj2.id) | obj1 is inside obj2 |
```

```
| BETWEEN | (obj1.id, obj2.id, obj3.id) | obj1 is between obj2 and
    obj3 |
| NEXT_TO | (obj1.id, obj2.id) | obj1 is close to or next to obj2 |
| FACING | (character1.id, obj1.id) | character1 is facing obj1 |
| HOLDS_RH | (character1.id, obj1.id) | character1 is holding obj1
    with right hand |
| HOLDS_LH | (character1.id, obj1.id) | character1 is holding obj1
    with left hand |
## Available Actions
Action primitive is similar to state primitive. Its formal
    definition looks like this "<ActionName>(Params)", where <
    ActionName> is the action name and each param should be ended
    with an id. Note that, you do not need to list actions in most
    cases. Only use actions when impossible to fully describe goal
    states with only allowable state primitives. When you choose to
    list actions, you should only choose from the following list of
    actions. For other cases, use state predicate as substitutes.
    Here, 'obj' only refers to items, not agents.
| Action Name | Arguments | Argument Restriction | Description |
| --- | --- | --- | --- |
| DRINK | (obj1.id) | obj1 is ['DRINKABLE', 'RECIPIENT'] | drinks
    obj1, need to hold obj1 first |
| EAT | (obj1.id) | obj1 is ['EATABLE'] | eats obj1, need to hold
    obj1 first |
| CUT | (obj1.id) | obj1 is ['EATABLE', 'CUTABLE'] | cuts obj1,
    obj1 is food|
| TOUCH | (obj1.id) | none | touches obj1 |
| LOOKAT | (obj1.id) | none | looks at obj1, it has a precondition
    that agent should be facing at obj1 first |
| WATCH | (obj1.id) | none | watches obj1 |
| READ | (obj1.id) | obj1 is ['READABLE'] | reads obj1, need to
    hold obj1 first |
| TYPE | (obj1.id) | obj1 is ['HAS_SWITCH'] | types on obj1 |
| PUSH | (obj1.id) | obj1 is ['MOVABLE'] | pushes obj1 |
| PULL | (obj1.id) | obj1 is ['MOVABLE'] | pulls obj1 |
| MOVE | (obj1.id) | obj1 is ['MOVABLE'] | moves obj1 |
| SQUEEZE | (obj1.id) | obj1 is ['CLOTHES'] | squeezes obj1 |
| SLEEP | none | none | sleeps, need to be at LYING or SITTING
    first |
| WAKEUP | none | none | wakes up, need to be at LYING or SITTING
    first |
| RINSE | (obj1.id) | none | rinses obj1, use only for cleaning
    appliances or teeth |
| SCRUB | (obj1.id) | none | scrubs obj1, use only for cleaning
    appliances or teeth |
| WASH | (obj1.id) | none | washes obj1, use only for appliances |
| GRAB | (obj1.id) | obj1 is ['GRABBABLE'] | grabs obj1 |
| SWITCHOFF | (obj1.id) | obj1 is ['HAS_SWITCH'] | switches off
    obj1 |
| POUR | (obj1.id, obj2.id) | none | pours obj1 into obj2 |

# Rules You Must Follow
- Temporal logic formula refers to a Boolean expression that
    describes a subgoals plan with temporal and logical order.
- The atomic Boolean expression includes both state primitive and
    action primitive.
- Boolean expressions in the same line are interchangeable with no
```

```
    temporal order requirement.
- Boolean expresssions in different lines are in temporal order,
    where the first one should be satisfied before the second one.
- Boolean expression can be combined with the following logical
    operators: "and", "or".
- The "and" operator combines Boolean expressions that are
    interchangeable but needs to be satisfied simultaneously in the
    end.
- The "or" operator combines Boolean expressions that are
    interchangeable but only one of them needs to be satisfied in
    the end.
- When there is temporal order requirement, output the Boolean
    expressions in different lines.
- Add intermediate states if necessary to improve logical
    consistency.
- If you want to change state of A, while A is in B and B is closed,
     you should make sure B is open first.
- Your output format should strictly follow this json format: {{"
    necessity_to_use_action": <necessity>, "actions_to_include": [<
    actions>], "output": [<your subgoal plan>]}}, where in <
    necessity> you should put "yes" or "no" to indicate whether
    actions should be included in subgoal plans. If you believe it
    is necessary to use actions, in the field <actions>, you should
    list all actions you used in your output. Otherwise, you should
    simply output an empty list []. In the field <your subgoal plan
    >, you should list all Boolean expressions in the required
    format and the temporal order.

Below are two examples for your better understanding.
## Example 1: Task category is "Listen to music"
## Relevant Objects in the Scene
| obj | category | properties |
| --- | --- | --- |
| bathroom.1 | Rooms | [] |
| character.65 | Characters | [] |
| home_office.319 | Rooms | [] |
| couch.352 | Furniture | ['LIEABLE', 'MOVABLE', 'SITTABLE', '
    SURFACES'] |
| television.410 | Electronics | ['HAS_PLUG', 'HAS_SWITCH', '
    LOOKABLE'] |
| dvd_player.1000 | placable_objects | ['CAN_OPEN', 'GRABBABLE', '
    HAS_PLUG', 'HAS_SWITCH', 'MOVABLE', 'SURFACES'] |

## Initial States
CLEAN(dvd_player.1000)
CLOSED(dvd_player.1000)
OFF(dvd_player.1000)
PLUGGED_IN(dvd_player.1000)
INSIDE(character.65, bathroom.1)

## Goal States
[States]
CLOSED(dvd_player.1000)
ON(dvd_player.1000)
PLUGGED_IN(dvd_player.1000)
[Actions Must Include]: Actions are listed in the execution order,
    each line is one action to satisfy. If "A or B or ..." is
```

```
    presented in one line, then only one of them needs to be
    satisfied.
None

## Necessity to Use Actions
No

## Output: Based on initial states in this task, achieve final goal
     states logically and reasonably. It does not matter which state
     should be satisfied first, as long as all goal states can be
    satisfied at the end. Make sure your output follows the json
    format, and do not include irrelevant information, do not
    include any explanation.
{{"necessity_to_use_action": "no", "actions_to_include": [], "
    output": ["NEXT_TO(character.65, dvd_player.1000)", "FACING(
    character.65, dvd_player.1000)", "PLUGGED_IN(dvd_player.1000)
    and CLOSED(dvd_player.1000)", "ON(dvd_player.1000)"]}}

# Example 2: Task category is "Browse internet"
## Relevant Objects in the Scene
| bathroom.1 | Rooms | [] |
| character.65 | Characters | [] |
| floor.208 | Floor | ['SURFACES'] |
| wall.213 | Walls | [] |
| home_office.319 | Rooms | [] |
| floor.325 | Floors | ['SURFACES'] |
| floor.326 | Floors | ['SURFACES'] |
| wall.330 | Walls | [] |
| wall.331 | Walls | [] |
| doorjamb.346 | Doors | [] |
| walllamp.351 | Lamps | [] |
| chair.356 | Furniture | ['GRABBABLE', 'MOVABLE', 'SITTABLE', '
    SURFACES'] |
| desk.357 | Furniture | ['MOVABLE', 'SURFACES'] |
| powersocket.412 | Electronics | [] |
| mouse.413 | Electronics | ['GRABBABLE', 'HAS_PLUG', 'MOVABLE'] |
| mousepad.414 | Electronics | ['MOVABLE', 'SURFACES'] |
| keyboard.415 | Electronics | ['GRABBABLE', 'HAS_PLUG', 'MOVABLE']
     |
| cpuscreen.416 | Electronics | [] |
| computer.417 | Electronics | ['HAS_SWITCH', 'LOOKABLE'] |

## Initial States
CLEAN(computer.417)
OFF(computer.417)
ONTOP(mouse.413, mousepad.414)
ONTOP(mouse.413, desk.357)
ONTOP(keyboard.415, desk.357)
INSIDE(character.65, bathroom.1)

## Goal States
[States]
ON(computer.417)
INSIDE(character.65, home_office.319)
HOLDS_LH(character.65, keyboard.415)
FACING(character.65, computer.417)
HOLDS_RH(character.65, mouse.413)
```

```
[Actions Must Include]: Actions are listed in the execution order,
    each line is one action to satisfy. If "A or B or ..." is
    presented in one line, then only one of them needs to be
    satisfied.
LOOKAT or WATCH

## Necessity to Use Actions
Yes

## Output: Based on initial states in this task, achieve final goal
     states logically and reasonably. It does not matter which state
     should be satisfied first, as long as all goal states can be
    satisfied at the end. Make sure your output follows the json
    format. Do not include irrelevant information, only output json
    object.
{{"necessity_to_use_action": "yes", "actions_to_include": ["LOOKAT
    "], "output": ["NEXT_TO(character.65, computer.417)", "ONTOP(
    character.65, chair.356)", "HOLDS_RH(character.65, mouse.413)
    and HOLDS_LH(character.65, keyboard.415)", "FACING(character.65,
     computer.417)", "LOOKAT(computer.417)"]}}

Now, it is time for you to generate the subgoal plan for the
    following task.
# Target Task: Task category is {task_name}
## Relevant Objects in the Scene
{relevant_objects}

## Initial States
{initial_states}

## Goal States
[States]
{final_states}
[Actions Must Include]: Actions are listed in the execution order,
    each line is one action to satisfy. If "A or B or ..." is
    presented in one line, then only one of them needs to be
    satisfied.
{final_actions}

## Necessity to Use Actions
{necessity}

## Output: Based on initial states in this task, achieve final goal
     states logically and reasonably. It does not matter which state
     should be satisfied first, as long as all goal states can be
    satisfied at the end. Make sure your output follows the json
    format. Do not include irrelevant information, only output json
    object.
```

Template 14: VirtualHome – Subgoal Decomposition Updated Template (v2)

```
# Background Introduction
You are an expert in symbolic AI and PDDL who is determining the
    complete state transitions of a household task for a assistive
    robot to assist a user agent. The objective is to derive an
```

```
    action plan to achieve the goal states from the initial states
    and list all intermediate states yields by each action (the
    effects of each action) in the plan to achieve the final target
    goals. These intermediate states are called subgoals. The state
    primitives include: unary primitives describing one object's own
     unary states and binary primitives describing object-object
    relationships. The output is a list of state and relation
    predicates constructed from the primitives detailed below.

More specifically, your task is to read and comprehend the input,
    reason about the world model with object states and
    relationships, and relevant actions for the goals. Then derive a
     sequence of actions needed to achieve the goals, but only
    provide the list of effects of those actions as a list of
    subgoals to achieve the goals in terms of symbolic predicates.
    The robot's inverse planner will take your output and attempt to
     plan for action sequences to iteratively achieve each subgoal.
    It will then base on this inferred goal of the user to derive
    its own assistive plan, but that is out of your scope. Thus,
    your list of subgoals needs to follow temporal logic order so
    success completions of subgoals can lead to achieving the final
    goal states. Sometimes the set of final goal states includes
    actions to be executed by the user. This means your plan needs
    to arrive at a set of final states that includes the
    precondition of these actions by the user agent. In such cases,
    your output will also need to include the action goal as part of
     the subgoal plan.

# State Primitives Vocabulary
Below we introduce the detailed data vocabulary and their format
    that you can use to generate the subgoal plan.
## Available States
State primitive is a tuple of a predicate name and its arguments.
    Its formal definition looks like this "<PredicateName>(Params)",
     where <PredicateName> is the state name and each param should
    be ended with an id. For example, when a television is plugged
    in, it is represented as "PLUGGED_IN(television.1). Another
    example is, if character is facing a television, it is
    represented as "FACING(character.1, television.1)". Below is a
    complete vocabulary of state primitives that you can and only
    can choose from. Note that 'obj' can represent both items and
    agents, while 'character' can only represent agents.
| Predicate Name | Arguments | Description |
| --- | --- | --- |
| CLOSED | (obj1.id) | obj1 is closed |
| OPEN | (obj1.id) | obj1 is open |
| ON | (obj1.id) | obj1 is turned on, or it is activated |
| OFF | (obj1.id) | obj1 is turned off, or it is deactivated |
| PLUGGED_IN | (obj1.id) | obj1 is plugged in |
| PLUGGED_OUT | (obj1.id) | obj1 is unplugged |
| SITTING | (character1.id) | character1 is sitting, and this
    represents a state of a character |
| LYING | (character1.id) | character1 is lying |
| CLEAN | (obj1.id) | obj1 is clean |
| DIRTY | (obj1.id) | obj1 is dirty |
| ONTOP | (obj1.id, obj2.id) | obj1 is on top of obj2 |
| INSIDE | (obj1.id, obj2.id) | obj1 is inside obj2 |
```

```
| BETWEEN | (obj1.id, obj2.id, obj3.id) | obj1 is between obj2 and
    obj3 |
| NEXT_TO | (obj1.id, obj2.id) | obj1 is next to or next to obj2 |
| FACING | (character1.id, obj1.id) | character1 is facing obj1 |
| HOLDS_RH | (character1.id, obj1.id) | character1 is holding obj1
    with right hand |
| HOLDS_LH | (character1.id, obj1.id) | character1 is holding obj1
    with left hand |
## Available Actions
Action primitive is similar to state primitive. Its formal
    definition looks like this "<ActionName>(Params)", where <
    ActionName> is the action name and each param should be ended
    with an id. Note that, you do not need to list actions in most
    cases, only when there are action goals. Here, 'obj' only refers
     to items, not agents. All actions are executed by the agents.
| Action Name | Arguments | Argument Restriction | Description |
| --- | --- | --- | --- |
| CLOSE | (obj1.id) | obj1 is ['OPENABLE'] | close sth |
| DRINK | (obj1.id) | obj1 is ['DRINKABLE', 'RECIPIENT'] | drink up
    sth |
| FIND | (obj1.id) | none | find and get next to sth |
| WALK | (obj1.id) | none | walk towards sth, get next to sth |
| GRAB | (obj1.id) | obj1 is ['GRABBABLE'] | grab sth |
| LOOKAT | (obj1.id) | none | look at sth, face sth |
| OPEN | (obj1.id) | obj1 is ['OPENABLE'] | open sth, as opposed to
    close sth |
| POINTAT | (obj1.id) | none | point at sth |
| PUTBACK | (obj1.id, obj2.id) | none | put object A back to object
    B |
| PUTIN | (obj1.id, obj2.id) | obj2 is ['CONTAINER'] | put object A
    into object B |
| RUN | (obj1.id) | none | run towards sth, get next to sth |
| SIT | (obj1.id) | obj1 is ['SITTABLE'] | sit on sth |
| STANDUP | none | none | stand up |
| SWITCHOFF | (obj1.id) | obj1 is ['HAS_SWITCH'] | switch sth off (
    normally lamp/light) |
| SWITCHON | (obj1.id) | obj1 is ['HAS_SWITCH'] | switch sth on (
    normally lamp/light) |
| TOUCH | (obj1.id) | none | touch sth |
| TURNTO | (obj1.id) | none | turn and face sth |
| WATCH | (obj1.id) | none | watch sth |
| WIPE | (obj1.id) | none | wipe sth out |
| PUTON | (obj1.id) | obj1 is ['CLOTHES'] | put on clothes, need to
    hold the clothes first |
| PUTOFF | (obj1.id) | obj1 is ['CLOTHES'] | put off clothes |
| GREET | (obj1.id) | obj1 is ['PERSON'] | greet to somebody |
| DROP | (obj1.id) | none | drop something in the current room,
    need to hold the thing first |
| READ | (obj1.id) | obj1 is ['READABLE'] | read something, need to
    hold the thing first |
| LIE | (obj1.id) | obj1 is ['LIEABLE'] | lie on something, need to
    get close the thing first |
| POUR | (obj1.id, obj2.id) | none | pour object A into object B |
| TYPE | (obj1.id) | obj1 is ['HAS_SWITCH', 'KEYBOARD'] | type on
    keyboard |
| PUSH | (obj1.id) | obj1 is ['MOVABLE'] | move sth, need to get
    next to sth first |
```

```
| PULL | (obj1.id) | obj1 is ['MOVABLE'] | move sth, need to get
    next to sth first |
| MOVE | (obj1.id) | obj1 is ['MOVABLE'] | move sth, need to get
    next to sth first |
| WASH | (obj1.id) | none | wash sth, need to get next to sth and
    hold cleaning tools first |
| RINSE | (obj1.id) | none | rinse sth, need to hold sth and next
    to a sink first |
| SCRUB | (obj1.id) | none | scrub sth, need to hold a brush and be
     next to sth first |
| SQUEEZE | (obj1.id) | obj1 is ['CLOTHES'] | squeeze the clothes |
| PLUGIN | (obj1.id) | obj1 is ['HAS_PLUG'] | plug in sth to a
    powersocket |
| PLUGOUT | (obj1.id) | obj1 is ['HAS_PLUG'] | plug out sth from a
    powersocket |
| CUT | (obj1.id) | obj1 is ['EATABLE', 'CUTABLE'] | cut some food
    |
| EAT | (obj1.id) | obj1 is ['EATABLE'] | eat some food |
| RELEASE | (obj1.id) | none | drop sth inside the current room |

# Rules You Must Follow
- Temporal logic formula refers to a Boolean expression that
    describes a subgoals plan with temporal and logical order.
- The initial states are the states that are given at the beginning
     of the task.
- Your content for the "output" key must be a list of boolean
    expressions that are in temporal order.
- The atomic Boolean expression includes both state primitive and
    action primitive.
- Boolean expressions in the same line are interchangeable with no
    temporal order requirement.
- Boolean expresssions in different lines are in temporal order,
    where the first one should be satisfied before the second one.
- Boolean expression can be combined with the following logical
    operators: "and", "or".
- The "and" operator combines Boolean expressions that are
    interchangeable but needs to be satisfied simultaneously in the
    end.
- The "or" operator combines Boolean expressions that are
    interchangeable but only one of them needs to be satisfied in
    the end.
- When there is temporal order requirement, output the Boolean
    expressions in different lines.
- The output must be consistent, logical and as detailed as
    possible. View your output as a complete state transition from
    the initial states to the final goal states.
- Please note that the agent can only hold one object in each hand.
     Also, the agent needs to have at least one hand free to perform
     any action other than DRINK, EAT, LOOKAT, WATCH, READ, PUSH,
    PULL, MOVE, SLEEP, WAKEUP, RINSE, SCRUB, WASH, POUR, RELEASE,
    DROP, LIE, RUN, STANDUP.
- Use HOLDS_RH and HOLDS_LH in your plan if necessary as they are
    objects' unary states. For example, DIRTY(shoe) cannot directly
    change to CLEAN(shoe), but needs (partial) intermediate states
    like [HOLDS_RH(brush), NEXT_TO(shoe), CLEAN(shoe)].
- Please use provided relevant objects well to help you achieve the
     final goal states.
```

```
- To make sure you output the correct results, reason about pseudo-
    actions necessary for state transitions and attempt planning for
     actions to traverse the subgoal sequence yourself to see of
    each subgoal and the final goal states are consistent and
    reachable from the initial states. You do not need to output
    this process though.
- Add intermediate states if necessary to improve logical
    consistency.
- Be very careful about minute itermediate subgoals such as open
    and close containers, etc. Those are VERY IMPORTANT subgoals.
- If you want to change state of A, while A is in B and B is closed,
     you should make sure B is open first.
- Your output format should strictly follow this json format: {{"
    necessity_to_use_action": <necessity>, "actions_to_include": [<
    actions>], "output": [<your subgoal plan>]}}, where in <
    necessity> you should put "yes" or "no" to indicate whether
    actions should be included in subgoal plans. If you believe it
    is necessary to use actions, in the field <actions>, you should
    list all actions you used in your output. Otherwise, you should
    simply output an empty list []. In the field <your subgoal plan
    >, you should list all Boolean expressions in the required
    format and the temporal order.

# SPECIAL RULES
* Due to a bug in the robot firmware, many kitchen and washing
    equipments can only take the binary state ONTOP but not the
    binary state INSIDE. For example, the robot can understand ONTOP
    (clothes_pants, washing_machine) but not INSIDE(clothes_pants,
    washing_machine). Some equipments sustaining this bug are:
    washing_machine , dishwasher , oven , coffe_maker . The only
    equipment that can work with INSIDE is "freezer".

# Input content
- The name of the target task can give hints on the priority of
    subgoals to be achieved in temporal order.
- Relevant objects in the scene indicates the potential involvement
     those objects in the sequence action executions required to
    achieve the final goal. Each object is listed in the form |
    object.id | Category | ['LIST', 'OF', 'SYMBOLIC', 'PROPERTIES']
    |
- Initial states are a set of state and relation predicates
    describing the initial status of each object in the scene.
- Goal states are the set of final symbolic goals we need to reach
    from the intial states.
- Actions must include are the target action goals we want the
    robot to assist. Action goals are listed in the execution order,
     each line is one action to satisfy. If "A or B or ..." is
    presented in one line, then only one of them needs to be
    executed by the user in the end.
- Necessity to use actions signifies if there are action goals. If
    necessity to use actions is no, the action goal should be None
    and be an empty list in the output.

# Input Format
## Target Task: <task_name>
## Relevant objects in this scene
<Objects_in_the_scene>
```

```
## Initial States
<initial_states>

## Goal States
[States]
<goal_states>
[Actions Must Include]
<action_goals>

## Necessity to use actions
<necessity>

# Output Format
- You must strictly output the following JSON format:{{"
    necessity_to_use_action": "<necessity>", "actions_to_include":
    [<action_goals>], "output": ["your", "subgoal", "plan"]}}
- Each element in the list are subgoals need to be achieved at each
     temporal stage in order to reach the final goal states. The
    subgoals in the preceding temporal stage needs to be satisfied
    before subgoals in the next temporal stage.
- For temporally interchangable subgoals, you can use connective '
    and' to combine them in the same temporal stage. This is
    optional and you are highly recommended to use a strict temporal
     order.
- If you think some states are equivalent, you can use connective '
    or' to combine them in the same temporal stage.
- The final goal states must be parts of your subgoal plan.
- The action goals must be parts of your subgoal plan.
- Do not output redundant states. A redundant state means a state
    that is either not necessary or has been satisfied before
    without taking any action.
- DO NOT INCLUDE IRRELEVANT INFORMATION (like number of line,
    explanation, etc.)

Below are two examples for your better understanding.
## Example 1: Task category is "Listen to music"
## Relevant Objects in the Scene
| obj | category | properties |
| --- | --- | --- |
| bathroom.1 | Rooms | [] |
| character.65 | Characters | [] |
| home_office.319 | Rooms | [] |
| couch.352 | Furniture | ['LIEABLE', 'MOVABLE', 'SITTABLE', '
    SURFACES'] |
| television.410 | Electronics | ['HAS_PLUG', 'HAS_SWITCH', '
    LOOKABLE'] |
| dvd_player.1000 | placable_objects | ['CAN_OPEN', 'GRABBABLE', '
    HAS_PLUG', 'HAS_SWITCH', 'MOVABLE', 'SURFACES'] |

## Initial States
CLEAN(dvd_player.1000)
CLOSED(dvd_player.1000)
OFF(dvd_player.1000)
PLUGGED_IN(dvd_player.1000)
INSIDE(character.65, bathroom.1)
```

```
## Goal States
[States]
CLOSED(dvd_player.1000)
ON(dvd_player.1000)
PLUGGED_IN(dvd_player.1000)
[Actions Must Include]:
None

## Necessity to Use Actions
No

## Output
{{"necessity_to_use_action": "no", "actions_to_include": [], "
    output": ["PLUGGED_IN(dvd_player.1000) and CLOSED(dvd_player
    .1000)", "NEXT_TO(character.65, dvd_player.1000)", "FACING(
    character.65, dvd_player.1000)", "ON(dvd_player.1000)"]}}

## Explanation (not part of the output):
The action plan for the example is
FIND(dvd_player.1000) -> effect: NEXT_TO(character.65, dvd_player
    .1000)
TURNTO(dvd_player.1000) -> effect: FACING(character.65, dvd_player
    .1000)
SWITCHON(dvd_player.1000) -> effect: ON(dvd_player.1000)
The initial states already satisfied parts of the goal states
    without actions for "PLUGGED_IN(dvd_player.1000) and CLOSED(
    dvd_player.1000)". Note that the agent was initially in the
    bathroom, where the DVD player is unlikely to be in, so we use
    FIND to get next to the DVD player in some other room.


# Example 2: Task category is "Browse internet"
## Relevant Objects in the Scene
| bathroom.1 | Rooms | [] |
| character.65 | Characters | [] |
| floor.208 | Floor | ['SURFACES'] |
| wall.213 | Walls | [] |
| home_office.319 | Rooms | [] |
| floor.325 | Floors | ['SURFACES'] |
| floor.326 | Floors | ['SURFACES'] |
| wall.330 | Walls | [] |
| wall.331 | Walls | [] |
| doorjamb.346 | Doors | [] |
| walllamp.351 | Lamps | [] |
| chair.356 | Furniture | ['GRABBABLE', 'MOVABLE', 'SITTABLE', '
    SURFACES'] |
| desk.357 | Furniture | ['MOVABLE', 'SURFACES'] |
| powersocket.412 | Electronics | [] |
| mouse.413 | Electronics | ['GRABBABLE', 'HAS_PLUG', 'MOVABLE'] |
| mousepad.414 | Electronics | ['MOVABLE', 'SURFACES'] |
| keyboard.415 | Electronics | ['GRABBABLE', 'HAS_PLUG', 'MOVABLE']
    |
| cpuscreen.416 | Electronics | [] |
| computer.417 | Electronics | ['HAS_SWITCH', 'LOOKABLE'] |

## Initial States
CLEAN(computer.417)
```

```
OFF(computer.417)
ONTOP(mouse.413, mousepad.414)
ONTOP(mouse.413, desk.357)
ONTOP(keyboard.415, desk.357)
INSIDE(character.65, bathroom.1)

## Goal States
[States]
ON(computer.417)
INSIDE(character.65, home_office.319)
HOLDS_LH(character.65, keyboard.415)
FACING(character.65, computer.417)
HOLDS_RH(character.65, mouse.413)
[Actions Must Include]:
LOOKAT or WATCH

## Necessity to Use Actions
Yes

## Output
{{"necessity_to_use_action": "yes", "actions_to_include": ["LOOKAT
    "], "output": [INSIDE(character.65, home_office.319), "NEXT_TO(
    character.65, keyboard.415)", "HOLDS_RH(character.65, mouse.413)
    ", "NEXT_TO(character.65, mouse.413)", "HOLDS_LH(character.65,
    keyboard.415)", "NEXT_TO(character.65, computer.417)", "ON(
    computer.417)", "NEXT_TO(character.65, chair.356)", "ONTOP(
    character.65, chair.356)", "FACING(character.65, computer.417)",
     "LOOKAT(computer.417)"]}}

## Explanation (not part of the output):
The action plan for the example is
WALK(home_office.319) -> effect: INSIDE(character.65, home_office)
FIND(keyboard.415) -> effect: NEXT_TO(character.65, keyboard.415)
GRAB(keyboard.415) -> effect: HOLDS_LH(character.65, keyboard.415)
FIND(mouse.413) -> effect: NEXT_TO(character.65, mouse.413)
GRAB(mouse.413) -> effect: HOLDS_RH(character.65, mouse.413)
FIND(computer.417) -> effect: NEXT_TO(character.65, computer.417)
SWITCHON(computer.417) -> effect: ON(computer.417)
FIND(chair.356) -> effect: NEXT_TO(character.65, chair.356)
SIT(chair.356) -> effect: ONTOP(character.65, chair.356)
TURNTO(computer.417) -> effect: FACING(character.65, computer.417)
LOOKAT(computer.417)

Now, it is time for you to generate the subgoal plan for the
    following task.

# Target Task: {task_name}
## Relevant Objects in the Scene
{relevant_objects}

## Initial States
{initial_states}

## Goal States
[States]
{final_states}
[Actions Must Include]:
```

```
{final_actions}

## Necessity to Use Actions
{necessity}

## Output:
```

D.3 Action Sequencing

D.3.1 BEHAVIOR

Template 15: BEHAVIOR – Action Sequencing Original Template

```
Problem:
You are designing instructions for a household robot.
The goal is to guide the robot to modify its environment from an
    initial state to a desired final state.
The input will be the initial environment state, the target
    environment state, the objects you can interact with in the
    environment.
The output should be a list of action commands so that after the
    robot executes the action commands sequentially, the environment
     will change from the initial state to the target state.

Data format: After # is the explanation.

Format of the states:
The environment state is a list starts with a uniary predicate or a
     binary prediate, followed by one or two obejcts.
You will be provided with multiple environment states as the
    initial state and the target state.
For example:
['inside', 'strawberry_0', 'fridge_97'] #strawberry_0 is inside
    fridge_97
['not', 'sliced', 'peach_0'] #peach_0 is not sliced
['ontop', 'jar_1', 'countertop_84'] #jar_1 is on top of
    countertop_84

Format of the action commands:
Action commands is a dictionary with the following format:
{{
      "action": "action_name",
      "object": "target_obj_name",
}}

or

{{
      "action": "action_name",
      "object": "target_obj_name1,target_obj_name2",
}}

The action_name must be one of the following:
```

```
LEFT_GRASP # the robot grasps the object with its left hand, to
    execute the action, the robot's left hand must be empty, e.g.
    {{'action': 'LEFT_GRASP', 'object': 'apple_0'}}.
RIGHT_GRASP # the robot grasps the object with its right hand, to
    execute the action, the robot's right hand must be empty, e.g.
    {{'action': 'RIGHT_GRASP', 'object': 'apple_0'}}.
LEFT_PLACE_ONTOP # the robot places the object in its left hand on
    top of the target object and release the object in its left hand
    , e.g. {{'action': 'LEFT_PLACE_ONTOP', 'object': 'table_1'}}.
RIGHT_PLACE_ONTOP # the robot places the object in its right hand
    on top of the target object and release the object in its left
    hand, e.g. {{'action': 'RIGHT_PLACE_ONTOP', 'object': 'table_1
    '}}.
LEFT_PLACE_INSIDE # the robot places the object in its left hand
    inside the target object and release the object in its left hand
    , to execute the action, the robot's left hand must hold an
    object, and the target object can't be closed e.g. {{'action': '
    LEFT_PLACE_INSIDE', 'object': 'fridge_1'}}.
RIGHT_PLACE_INSIDE # the robot places the object in its right hand
    inside the target object and release the object in its left hand
    , to execute the action, the robot's right hand must hold an
    object, and the target object can't be closed, e.g. {{'action':
    'RIGHT_PLACE_INSIDE', 'object': 'fridge_1'}}.
RIGHT_RELEASE # the robot directly releases the object in its right
     hand, to execute the action, the robot's left hand must hold an
     object, e.g. {{'action': 'RIGHT_RELEASE', 'object': 'apple_0
    '}}.
LEFT_RELEASE # the robot directly releases the object in its left
    hand, to execute the action, the robot's right hand must hold an
     object, e.g. {{'action': 'LEFT_RELEASE', 'object': 'apple_0'}}.
OPEN # the robot opens the target object, to execute the action,
    the target object should be openable and closed, also, toggle
    off the target object first if want to open it, e.g. {{'action':
     'OPEN', 'object': 'fridge_1'}}.
CLOSE # the robot closes the target object, to execute the action,
    the target object should be openable and open, e.g. {{'action':
    'CLOSE', 'object': 'fridge_1'}}.
COOK # the robot cooks the target object, to execute the action,
    the target object should be put in a pan, e.g. {{'action': 'COOK
    ', 'object': 'apple_0'}}.
CLEAN # the robot cleans the target object, to execute the action,
    the robot should have a cleaning tool such as rag, the cleaning
    tool should be soaked if possible, or the target object should
    be put into a toggled on cleaner like a sink or a dishwasher, e.
    g. {{'action': 'CLEAN', 'object': 'window_0'}}.
FREEZE # the robot freezes the target object e.g. {{'action': '
    FREEZE', 'object': 'apple_0'}}.
UNFREEZE # the robot unfreezes the target object, e.g. {{'action': '
    UNFREEZE', 'object': 'apple_0'}}.
SLICE # the robot slices the target object, to execute the action,
    the robot should have a knife in hand, e.g. {{'action': 'SLICE',
     'object': 'apple_0'}}.
SOAK # the robot soaks the target object, to execute the action,
    the target object must be put in a toggled on sink, e.g. {{'
    action': 'SOAK', 'object': 'rag_0'}}.
DRY # the robot dries the target object, e.g. {{'action': 'DRY', '
    object': 'rag_0'}}.
```

```
TOGGLE_ON # the robot toggles on the target object, to execute the
    action, the target object must be closed if the target object is
     openable and open e.g. {{'action': 'TOGGLE_ON', 'object': '
    light_0'}}.
TOGGLE_OFF # the robot toggles off the target object, e.g. {{'
    action': 'TOGGLE_OFF', 'object': 'light_0'}}.
LEFT_PLACE_NEXTTO # the robot places the object in its left hand
    next to the target object and release the object in its left
    hand, e.g. {{'action': 'LEFT_PLACE_NEXTTO', 'object': 'table_1
    '}}.
RIGHT_PLACE_NEXTTO # the robot places the object in its right hand
    next to the target object and release the object in its right
    hand, e.g. {{'action': 'RIGHT_PLACE_NEXTTO', 'object': 'table_1
    '}}.
LEFT_TRANSFER_CONTENTS_INSIDE # the robot transfers the contents in
     the object in its left hand inside the target object, e.g. {{'
    action': 'LEFT_TRANSFER_CONTENTS_INSIDE', 'object': 'bow_1'}}.
RIGHT_TRANSFER_CONTENTS_INSIDE # the robot transfers the contents
    in the object in its right hand inside the target object, e.g.
    {{'action': 'RIGHT_TRANSFER_CONTENTS_INSIDE', 'object': 'bow_1
    '}}.
LEFT_TRANSFER_CONTENTS_ONTOP # the robot transfers the contents in
    the object in its left hand on top of the target object, e.g.
    {{'action': 'LEFT_TRANSFER_CONTENTS_ONTOP', 'object': 'table_1
    '}}.
RIGHT_TRANSFER_CONTENTS_ONTOP # the robot transfers the contents in
     the object in its right hand on top of the target object, e.g.
    {{'action': 'RIGHT_TRANSFER_CONTENTS_ONTOP', 'object': 'table_1
    '}}.
LEFT_PLACE_NEXTTO_ONTOP # the robot places the object in its left
    hand next to target object 1 and on top of the target object 2
    and release the object in its left hand, e.g. {{'action': '
    LEFT_PLACE_NEXTTO_ONTOP', 'object': 'window_0, table_1'}}.
RIGHT_PLACE_NEXTTO_ONTOP # the robot places the object in its right
     hand next to object 1 and on top of the target object 2 and
    release the object in its right hand, e.g. {{'action': '
    RIGHT_PLACE_NEXTTO_ONTOP', 'object': 'window_0, table_1'}}.
LEFT_PLACE_UNDER # the robot places the object in its left hand
    under the target object and release the object in its left hand,
     e.g. {{'action': 'LEFT_PLACE_UNDER', 'object': 'table_1'}}.
RIGHT_PLACE_UNDER # the robot places the object in its right hand
    under the target object and release the object in its right hand
    , e.g. {{'action': 'RIGHT_PLACE_UNDER', 'object': 'table_1'}}.

Format of the interactable objects:
Interactable object will contain multiple lines, each line is a
    dictionary with the following format:
{{
   "name": "object_name",
   "category": "object_category"
}}
object_name is the name of the object, which you must use in the
    action command, object_category is the category of the object,
    which provides a hint for you in interpreting initial and goal
    condtions.

Please pay specail attention:
```

```
1. The robot can only hold one object in each hand.
2. Action name must be one of the above action names, and the
    object name must be one of the object names listed in the
    interactable objects.
3. All PLACE actions will release the object in the robot's hand,
    you don't need to explicitly RELEASE the object after the PLACE
    action.
4. For LEFT_PLACE_NEXTTO_ONTOP and RIGHT_PLACE_NEXTTO_ONTOP, the
    action command are in the format of {{'action': 'action_name', '
    object': 'obj_name1, obj_name2'}}
5. If you want to perform an action to an target object, you must
    make sure the target object is not inside a closed object.
6. For actions like OPEN, CLOSE, SLICE, COOK, CLEAN, SOAK, DRY,
    FREEZE, UNFREEZE, TOGGLE_ON, TOGGLE_OFF, at least one of the
    robot's hands must be empty, and the target object must have the
     corresponding property like they're openable, toggleable, etc.
7. For PLACE actions and RELEASE actions, the robot must hold an
    object in the corresponding hand.
8. Before slicing an object, the robot can only interact with the
    object (e.g. peach_0), after slicing the object, the robot can
    only interact with the sliced object (e.g. peach_0_part_0).


Examples: after# is the explanation.

Example 1:
Input:
initial environment state:
['stained', 'sink_7']
['stained', 'bathtub_4']
['not', 'soaked', 'rag_0']
['onfloor', 'rag_0', 'room_floor_bathroom_0']
['inside', 'rag_0', 'cabinet_1']
['not', 'open', 'cabinet_1']


target environment state:
['not', 'stained', 'bathtub_4']
['not', 'stained', 'sink_7']
['and', 'soaked', 'rag_0', 'inside', 'rag_0', 'bucket_0']


interactable objects:
{{'name': 'sink_7', 'category': 'sink.n.01'}}
{{'name': 'bathtub_4', 'category': 'bathtub.n.01'}}
{{'name': 'bucket_0', 'category': 'bucket.n.01'}}
{{'name': 'rag_0', 'category': 'rag.n.01'}}
{{'name': 'cabinet_1', 'category': 'cabinet.n.01'}}


Please output the list of action commands (in the given format) so
    that after the robot executes the action commands sequentially,
    the current environment state will change to target environment
    state. Usually, the robot needs to execute multiple action
    commands consecutively to achieve final state. Please output
    multiple action commands rather than just one. Only output the
    list of action commands with nothing else.
```

```
Output:
[
   {{
      "action": "OPEN",
      "object": "cabinet_1"
   }}, # you want to get the rag_0 from cabinet_1, should open it
       first
   {{
      "action": "RIGHT_GRASP",
      "object": "rag_0"
   }}, # you want to clean the sink_7 and bathtub_4, you found them
        stained, so you need to soak the rag_0 first
   {{
      "action": "RIGHT_PLACE_INSIDE",
      "object": "sink_7"
   }}, # to soak the rag_0, you need to place it inside the sink_7
   {{
      "action": "TOGGLE_ON",
      "object": "sink_7"
   }}, # to soak the rag_0, you need to toggle on the sink_7
   {{
      "action": "SOAK",
      "object": "rag_0"
   }}, # now you can soak the rag_0
   {{
      "action": "TOGGLE_OFF",
      "object": "sink_7"
   }}, # after soaking the rag_0, you need to toggle off the sink_7
   {{
      "action": "LEFT_GRASP",
      "object": "rag_0"
   }}, # now you can grasp soaked rag_0 to clean stain
   {{
      "action": "CLEAN",
      "object": "sink_7"
   }}, # now you clean the sink_7
   {{
      "action": "CLEAN",
      "object": "bathtub_4"
   }}, # now you clean the bathtub_4
   {{
      "action": "LEFT_PLACE_INSIDE",
      "object": "bucket_0"
   }} # after cleaning the sink_7, you need to place the rag_0
       inside the bucket_0
]

Your task:
Input:
initial environment state:
{init_state}

target environment state:
{target_state}

interactable objects:
```

```
{obj_list}

Please output the list of action commands (in the given format) so
    that after the robot executes the action commands sequentially,
    the current environment state will change to target environment
    state. Usually, the robot needs to execute multiple action
    commands consecutively to achieve final state. Please output
    multiple action commands rather than just one. Only output the
    list of action commands with nothing else.

Output:
```

Template 16: BEHAVIOR – Action Sequencing PDDL Template

```
# Background
You are an expert PDDL (Planning Domain Definition Language)
    analyzer and debugger. You are given the following PDDL domain
    file:

---
[PDDL Domain File]
(define (domain igibson)

   (:requirements :strips :adl :typing :negative-preconditions)

   (:types
      vacuum_n_04 facsimile_n_02 dishtowel_n_01 apparel_n_01
          seat_n_03 bottle_n_01 mouse_n_04 window_n_01 scanner_n_02
      sauce_n_01 spoon_n_01 date_n_08 egg_n_02 cabinet_n_01
          yogurt_n_01 parsley_n_02 notebook_n_01 dryer_n_01
          saucepan_n_01
      soap_n_01 package_n_02 headset_n_01 fish_n_02 vehicle_n_01
          chestnut_n_03 grape_n_01 wrapping_n_01 makeup_n_01
          mug_n_04
      pasta_n_02 beef_n_02 scrub_brush_n_01 cracker_n_01 flour_n_01
           sunglass_n_01 cookie_n_01 bed_n_01 lamp_n_02 food_n_02
      painting_n_01 carving_knife_n_01 pop_n_02 tea_bag_n_01
          sheet_n_03 tomato_n_01 agent_n_01 hat_n_01 dish_n_01
          cheese_n_01
      perfume_n_02 toilet_n_02 broccoli_n_02 book_n_02 towel_n_01
          table_n_02 pencil_n_01 rag_n_01 peach_n_03 water_n_06
          cup_n_01
      radish_n_01 marker_n_03 tile_n_01 box_n_01 screwdriver_n_01
          raspberry_n_02 banana_n_02 grill_n_02 caldron_n_01
          vegetable_oil_n_01
      necklace_n_01 brush_n_02 washer_n_03 hamburger_n_01
          catsup_n_01 sandwich_n_01 plaything_n_01 candy_n_01
          cereal_n_03 door_n_01
      food_n_01 newspaper_n_03 hanger_n_02 carrot_n_03 salad_n_01
          toothpaste_n_01 blender_n_01 sofa_n_01 plywood_n_01
          olive_n_04 briefcase_n_01
      christmas_tree_n_05 bowl_n_01 casserole_n_02 apple_n_01
          basket_n_01 pot_plant_n_01 backpack_n_01 sushi_n_01
          saw_n_02 toothbrush_n_01
      lemon_n_01 pad_n_01 receptacle_n_01 sink_n_01 countertop_n_01
```

```
        melon_n_01 bracelet_n_02 modem_n_01 pan_n_01 oatmeal_n_01
        calculator_n_02
    duffel_bag_n_01 sandal_n_01 floor_n_01 snack_food_n_01
        stocking_n_01 dishwasher_n_01 pencil_box_n_01 chicken_n_01
        jar_n_01 alarm_n_02
    stove_n_01 plate_n_04 highlighter_n_02 umbrella_n_01
        piece_of_cloth_n_01 bin_n_01 ribbon_n_01 chip_n_04
        shelf_n_01 bucket_n_01 shampoo_n_01
    folder_n_02 shoe_n_01 detergent_n_02 milk_n_01 beer_n_01
        shirt_n_01 dustpan_n_02 cube_n_05 broom_n_01 candle_n_01
        pen_n_01 microwave_n_02
    knife_n_01 wreath_n_01 car_n_01 soup_n_01 sweater_n_01
        tray_n_01 juice_n_01 underwear_n_01 orange_n_01
        envelope_n_01 fork_n_01 lettuce_n_03
    bathtub_n_01 earphone_n_01 pool_n_01 printer_n_03 sack_n_01
        highchair_n_01 cleansing_agent_n_01 kettle_n_01
        vidalia_onion_n_01 mousetrap_n_01
    bread_n_01 meat_n_01 mushroom_n_05 cake_n_03 vessel_n_03
        bow_n_08 gym_shoe_n_01 hammer_n_02 teapot_n_01 chair_n_01
        jewelry_n_01 pumpkin_n_02 sugar_n_01
    shower_n_01 ashcan_n_01 hand_towel_n_01 pork_n_01
        strawberry_n_01 electric_refrigerator_n_01 oven_n_01
        ball_n_01 document_n_01 sock_n_01 beverage_n_01
    hardback_n_01 scraper_n_01 carton_n_02
    agent
)

(:predicates
    (frozen ?obj1 - object)
    (inside ?obj1 - object ?obj2 - object)
    (nextto ?obj1 - object ?obj2 - object)
    (touching ?agent1 - agent ?obj2 - object)
    (dusty ?obj1 - object)
    (soaked ?obj1 - object)
    (under ?obj1 - object ?obj2 - object)
    (toggled_on ?obj1 - object)
    (sliced ?obj1 - object)
    (burnt ?obj1 - object)
    (ontop ?obj1 - object ?obj2 - object)
    (cooked ?obj1 - object)
    (onfloor ?obj1 - object ?floor1 - object)
    (stained ?obj1 - object)
    (open ?obj1 - object)
    (holds_rh ?obj1 - object)
    (holds_lh ?obj1 - object)
    ; (holding ?obj1 - object)
    ; (handsfull ?agent1 - agent)
    (in_reach_of_agent ?obj1 - object)
    (same_obj ?obj1 - object ?obj2 - object)
)

(:action navigate_to
    :parameters (?objto - object ?agent - agent)
    :precondition (not (in_reach_of_agent ?objto))
    :effect (and (in_reach_of_agent ?objto)
             (forall
                (?objfrom - object)
```

```
                (when
                   (and
                      (in_reach_of_agent ?objfrom)
                      (not (same_obj ?objfrom ?objto))
                   )
                   (not (in_reach_of_agent ?objfrom))
                )
             )
          )
    )

    (:action left_grasp
    :parameters (?obj - object ?agent - agent)
    :precondition (and (not (exists (?floor - floor_n_01) (same_obj
        ?obj ?floor)))
                (not (exists (?fridge -
                    electric_refrigerator_n_01) (same_obj ?obj ?
                    fridge)))
                (not (exists (?sink - sink_n_01) (same_obj ?obj ?
                    sink)))
                (not (exists (?dishwasher - dishwasher_n_01) (
                    same_obj ?obj ?dishwasher)))
                (not (exists (?table - table_n_02) (same_obj ?obj
                     ?table)))
                (not (exists (?stove - stove_n_01) (same_obj ?obj
                     ?stove)))
                (not (exists (?oven - oven_n_01) (same_obj ?obj ?
                    oven)))
                (not (exists (?car - car_n_01) (same_obj ?obj ?
                    car)))
                (not (exists (?pool - pool_n_01) (same_obj ?obj ?
                    pool)))
                (not (exists (?bathtub - bathtub_n_01) (same_obj
                    ?obj ?bathtub)))
                (not (exists (?shelf - shelf_n_01) (same_obj ?obj
                     ?shelf)))
                (not (exists (?bed - bed_n_01) (same_obj ?obj ?
                    bed)))
                (not (exists (?cabinet - cabinet_n_01) (same_obj
                    ?obj ?cabinet)))
                (not (exists (?obj2 - object) (holds_lh ?obj2)))
                (not (exists (?obj3 - object) (and (holds_rh ?
                    obj3) (same_obj ?obj ?obj3))))
                (in_reach_of_agent ?obj)
                (not (exists (?obj2 - object) (and (inside ?obj ?
                    obj2) (not (open ?obj2)))))
             )
    :effect (and (holds_lh ?obj)
             ;; Conditional effects for all predicates involving
                 ?obj and ?other_obj
             (forall (?other_obj - object)
                (and (not (inside ?obj ?other_obj))
                    (not (ontop ?obj ?other_obj))
                    (not (under ?obj ?other_obj))
                    (not (under ?other_obj ?obj))
                    (not (nextto ?obj ?other_obj))
                    (not (nextto ?other_obj ?obj))
```

```
                    (not (onfloor ?obj ?other_obj))
                    ;; Add other predicates as needed
                )
            )
            (forall (?other_obj - object)
                (when (inside ?other_obj ?obj)
                    (holds_lh ?other_obj)
                )
            )
        )
    )

    (:action right_grasp
    :parameters (?obj - object ?agent - agent)
    :precondition (and (not (exists (?floor - floor_n_01) (same_obj
        ?obj ?floor)))
                (not (exists (?fridge -
                    electric_refrigerator_n_01) (same_obj ?obj ?
                    fridge)))
                (not (exists (?sink - sink_n_01) (same_obj ?obj ?
                    sink)))
                (not (exists (?dishwasher - dishwasher_n_01) (
                    same_obj ?obj ?dishwasher)))
                (not (exists (?table - table_n_02) (same_obj ?obj
                     ?table)))
                (not (exists (?stove - stove_n_01) (same_obj ?obj
                     ?stove)))
                (not (exists (?oven - oven_n_01) (same_obj ?obj ?
                    oven)))
                (not (exists (?car - car_n_01) (same_obj ?obj ?
                    car)))
                (not (exists (?pool - pool_n_01) (same_obj ?obj ?
                    pool)))
                (not (exists (?bathtub - bathtub_n_01) (same_obj
                    ?obj ?bathtub)))
                (not (exists (?shelf - shelf_n_01) (same_obj ?obj
                     ?shelf)))
                (not (exists (?bed - bed_n_01) (same_obj ?obj ?
                    bed)))
                (not (exists (?cabinet - cabinet_n_01) (same_obj
                    ?obj ?cabinet)))
                (not (exists (?obj2 - object) (holds_rh ?obj2)))
                (not (exists (?obj3 - object) (and (holds_lh ?
                    obj3) (same_obj ?obj ?obj3))))
                (in_reach_of_agent ?obj)
                (not (exists (?obj2 - object) (and (inside ?obj ?
                    obj2) (not (open ?obj2)))))
            )
    :effect (and (holds_rh ?obj)
            ;; Conditional effects for all predicates involving
                ?obj and ?other_obj
            (forall (?other_obj - object)
                (and (not (inside ?obj ?other_obj))
                    (not (ontop ?obj ?other_obj))
                    (not (under ?obj ?other_obj))
                    (not (under ?other_obj ?obj))
                    (not (nextto ?obj ?other_obj))
```

```
                    (not (nextto ?other_obj ?obj))
                    (not (onfloor ?obj ?other_obj))
                    ;; Add other predicates as needed
                )
            )
            (forall (?other_obj - object)
                (when (inside ?other_obj ?obj)
                    (holds_rh ?other_obj)
                )
            )
        )
)

(:action left_place_ontop
    :parameters (?obj_in_hand - object ?obj - object ?agent -
        agent)
    :precondition (and (holds_lh ?obj_in_hand)
                    (in_reach_of_agent ?obj)
                    (not (holds_rh ?obj))
    )
    :effect (and (ontop ?obj_in_hand ?obj)
                (not (holds_lh ?obj_in_hand))
                (forall (?other_obj - object)
                    (when (inside ?other_obj ?obj)
                        (not (holds_lh ?other_obj))
                    )
                )
    )
)

(:action right_place_ontop
    :parameters (?obj_in_hand - object ?obj - object ?agent -
        agent)
    :precondition (and (not (holds_lh ?obj))
                    (holds_rh ?obj_in_hand)
                    (in_reach_of_agent ?obj)
    )
    :effect (and (ontop ?obj_in_hand ?obj)
                (not (holds_rh ?obj_in_hand))
                (forall (?other_obj - object)
                    (when (inside ?other_obj ?obj)
                        (not (holds_rh ?other_obj))
                    )
                )
    )
)

(:action left_place_inside
    :parameters (?obj_in_hand - object ?obj - object ?agent -
        agent)
    :precondition (and (not (holds_rh ?obj))
                    (holds_lh ?obj_in_hand)
                    (in_reach_of_agent ?obj)
                    (open ?obj)
    )
    :effect (and (inside ?obj_in_hand ?obj)
                (not (holds_lh ?obj_in_hand))
```

```
                (forall (?other_obj - object) (not (holds_lh ?
                    other_obj)))
                (forall (?inside_obj - object)
                   (when (inside ?inside_obj ?obj_in_hand)
                      (inside ?inside_obj ?obj)
                   )
                )
          )
    )

    (:action right_place_inside
       :parameters (?obj_in_hand - object ?obj - object ?agent -
           agent)
       :precondition (and (not (holds_lh ?obj))
                     (holds_rh ?obj_in_hand)
                     (in_reach_of_agent ?obj)
                     (open ?obj)
       )
       :effect (and (inside ?obj_in_hand ?obj)
                (not (holds_rh ?obj_in_hand))
                (forall (?other_obj - object) (not (holds_rh ?
                    other_obj)))
                (forall (?inside_obj - object)
                   (when (inside ?inside_obj ?obj_in_hand)
                      (inside ?inside_obj ?obj)
                   )
                )
       )
    )

    (:action right_release
       :parameters (?obj - object ?agent - agent)
       :precondition (and (holds_rh ?obj))
       :effect (and (not (holds_rh ?obj))
                (forall (?other_obj - object) (not (holds_rh ?
                    other_obj)))
          )
    )

    (:action left_release
       :parameters (?obj - object ?agent - agent)
       :precondition (and (holds_lh ?obj))
       :effect (and (not (holds_lh ?obj))
                (forall (?other_obj - object) (not (holds_lh ?
                    other_obj)))
          )
    )

    (:action open
       :parameters (?obj - object ?agent - agent)
       :precondition (and (in_reach_of_agent ?obj)
                     (not (open ?obj))
                     (not (and (exists (?other_obj - object) (
                         holds_lh ?other_obj))
                           (exists (?other_obj - object) (holds_rh
                               ?other_obj))
                        )
```

```
                    )
                    (not
                       (or
                          (exists (?sink - sink_n_01) (same_obj ?
                             obj ?sink))
                          (exists (?stove - stove_n_01) (same_obj
                             ?obj ?stove))
                          (exists (?saucepan - saucepan_n_01) (
                             same_obj ?obj ?saucepan))
                          (exists (?bucket - bucket_n_01) (
                             same_obj ?obj ?bucket))
                          (exists (?bathtub - bathtub_n_01) (
                             same_obj ?obj ?bathtub))
                          (exists (?casserole - casserole_n_02) (
                             same_obj ?obj ?casserole))
                          (exists (?dish - dish_n_01) (same_obj ?
                             obj ?dish))
                          (exists (?bowl - bowl_n_01) (same_obj ?
                             obj ?bowl))
                          (exists (?carton - carton_n_02) (
                             same_obj ?obj ?carton))
                          (exists (?ashcan - ashcan_n_01) (
                             same_obj ?obj ?ashcan))
                          (exists (?shelf - shelf_n_01) (same_obj
                             ?obj ?shelf))
                          (exists (?basket - basket_n_01) (
                             same_obj ?obj ?basket))
                       )
                    )
              )
   :effect (open ?obj)
)

(:action open_skip
   :parameters (?obj - object ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                    (not (open ?obj))
                    (or
                       (exists (?sink - sink_n_01) (same_obj ?obj
                          ?sink))
                       (exists (?stove - stove_n_01) (same_obj ?
                          obj ?stove))
                       (exists (?saucepan - saucepan_n_01) (
                          same_obj ?obj ?saucepan))
                       (exists (?bucket - bucket_n_01) (same_obj ?
                          obj ?bucket))
                       (exists (?bathtub - bathtub_n_01) (same_obj
                           ?obj ?bathtub))
                       (exists (?casserole - casserole_n_02) (
                          same_obj ?obj ?casserole))
                       (exists (?dish - dish_n_01) (same_obj ?obj
                          ?dish))
                       (exists (?bowl - bowl_n_01) (same_obj ?obj
                          ?bowl))
                       (exists (?carton - carton_n_02) (same_obj ?
                          obj ?carton))
                       (exists (?ashcan - ashcan_n_01) (same_obj ?
```

```
                        obj ?ashcan))
                     (exists (?shelf - shelf_n_01) (same_obj ?
                        obj ?shelf))
                     (exists (?basket - basket_n_01) (same_obj ?
                        obj ?basket))
                  )
               )
   :effect (open ?obj)
)

(:action close
   :parameters (?obj - object ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (open ?obj)
                  (not (and (exists (?other_obj - object) (
                      holds_lh ?other_obj))
                     (exists (?other_obj - object) (holds_rh
                         ?other_obj))
                     )
                  )
               )
   :effect (not (open ?obj))
)

(:action cook
   :parameters (?obj - object ?agent - agent)
   :precondition (and (not (cooked ?obj))
                (exists (?pan - saucepan_n_01)
                   (and (inside ?obj ?pan)
                      (in_reach_of_agent ?pan)
                   )
                )
   )
   :effect (cooked ?obj)
)

(:action clean_dust_and_stain
   :parameters (?obj - object ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (or
                     (stained ?obj)
                     (dusty ?obj)
                  )
                  (or
                     (exists (?scrub_brush - scrub_brush_n_01)
                        (and (or (exists (?detergent -
                            detergent_n_02) (or (holds_lh ?
                            detergent) (holds_rh ?detergent)))
                               (soaked ?scrub_brush)
                              )
                              (or (holds_lh ?scrub_brush) (holds_rh
                                   ?scrub_brush))
                        )
                     )
                     (exists (?cloth - piece_of_cloth_n_01)
                        (and (or (exists (?detergent -
                            detergent_n_02) (or (holds_lh ?
```

```
                              detergent) (holds_rh ?detergent)))
                                 (soaked ?cloth)
                              )
                              (or (holds_lh ?cloth) (holds_rh ?
                                 cloth))
                           )
                        )
                        (exists (?hand_towel - hand_towel_n_01)
                           (and (or (exists (?detergent -
                              detergent_n_02) (or (holds_lh ?
                              detergent) (holds_rh ?detergent)))
                                 (soaked ?hand_towel)
                              )
                              (or (holds_lh ?hand_towel) (holds_rh
                                 ?hand_towel))
                           )
                        )
                        (exists (?towel - towel_n_01)
                           (and (or (exists (?detergent -
                              detergent_n_02) (or (holds_lh ?
                              detergent) (holds_rh ?detergent)))
                                 (soaked ?towel)
                              )
                              (or (holds_lh ?towel) (holds_rh ?
                                 towel))
                           )
                        )
                        (exists (?dishtowel - dishtowel_n_01)
                           (and (or (exists (?detergent -
                              detergent_n_02) (or (holds_lh ?
                              detergent) (holds_rh ?detergent)))
                                 (soaked ?dishtowel)
                              )
                              (or (holds_lh ?dishtowel) (holds_rh ?
                                 dishtowel))
                           )
                        )
                        (exists (?rag - rag_n_01)
                           (and (or (exists (?detergent -
                              detergent_n_02) (or (holds_lh ?
                              detergent) (holds_rh ?detergent)))
                                 (soaked ?rag)
                              )
                              (or (holds_lh ?rag) (holds_rh ?rag))
                           )
                        )
                        ; (exists (?dishwasher - dishwasher_n_01)
                        ; (and (inside ?obj ?dishwasher)
                        ; (toggled_on ?dishwasher))
                        ; )
                        (exists (?sink - sink_n_01)
                           (and (inside ?obj ?sink)
                              (toggled_on ?sink))
                        )
                     )
      )
      :effect (and (not (dusty ?obj)) (not (stained ?obj)))
```

```
)

(:action clean_dust
   :parameters (?obj - object ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (dusty ?obj)
                  (or
                     (exists (?scrub_brush - scrub_brush_n_01)
                        (or (holds_lh ?scrub_brush) (holds_rh ?
                           scrub_brush))
                     )
                     (exists (?cloth - piece_of_cloth_n_01)
                        (or (holds_lh ?cloth) (holds_rh ?cloth))
                     )
                     (exists (?hand_towel - hand_towel_n_01)
                        (or (holds_lh ?hand_towel) (holds_rh ?
                           hand_towel))
                     )
                     (exists (?towel - towel_n_01)
                        (or (holds_lh ?towel) (holds_rh ?towel))
                     )
                     (exists (?dishtowel - dishtowel_n_01)
                        (or (holds_lh ?dishtowel) (holds_rh ?
                           dishtowel))
                     )
                     (exists (?rag - rag_n_01)
                        (or (holds_lh ?rag) (holds_rh ?rag))
                     )
                     (exists (?dishwasher - dishwasher_n_01)
                        (and (inside ?obj ?dishwasher)
                           (toggled_on ?dishwasher))
                     )
                  )
   )
   :effect (not (dusty ?obj))
)

(:action freeze
   :parameters (?obj - object ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (not (frozen ?obj)))
   :effect (frozen ?obj)
)

(:action unfreeze
   :parameters (?obj - object ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (frozen ?obj))
   :effect (not (frozen ?obj))
)

(:action slice
   :parameters (?obj - object ?agent - agent)
   :precondition (and (or
                     (exists (?knife - knife_n_01)
                        (or (holds_lh ?knife) (holds_rh ?knife)))
```

```
                        (exists (?carving_knife -
                            carving_knife_n_01)
                           (or (holds_lh ?carving_knife) (holds_rh
                               ?carving_knife)))
                     )
                     ; (exists (?board - countertop_n_01) (ontop ?
                         obj ?board))
                     (in_reach_of_agent ?obj)
                     (not (sliced ?obj))
      )
      :effect (sliced ?obj)
   )

   (:action soak
      :parameters (?obj1 - object ?agent - agent)
      :precondition (and (exists (?sink - sink_n_01)
                     (and
                        (in_reach_of_agent ?sink)
                        (toggled_on ?sink)
                        (inside ?obj1 ?sink)
                     ))
                     (not (soaked ?obj1))
                     ; (or (holds_lh ?obj1) (holds_rh ?obj1))
      )
      :effect (soaked ?obj1)
   )

   (:action soak_teapot
      :parameters (?teapot - teapot_n_01 ?obj1 - object ?agent -
          agent)
      :precondition (and (or (holds_lh ?obj1) (holds_rh ?obj1))
                     (in_reach_of_agent ?teapot)
                     (inside ?obj1 ?teapot)
                     (not (soaked ?obj1))
      )
      :effect (soaked ?obj1)
   )

   (:action dry
      :parameters (?obj - object ?agent - agent)
      :precondition (and (in_reach_of_agent ?obj1)
                  ; (holds_lh ?obj1)
                  ; (holds_rh ?obj1)
                  (soaked ?obj1)
      )
      :effect (not (soaked ?obj1))
   )

   (:action toggle_on
      :parameters (?obj - object ?agent - agent)
      :precondition (and (in_reach_of_agent ?obj)
                     (not
                        (and
                           (exists (?other_obj - object) (holds_lh
                               ?other_obj))
                           (exists (?other_obj - object) (holds_rh
                               ?other_obj))
```

```
                        )
                    )
                    (not (toggled_on ?obj))
        )
        :effect (toggled_on ?obj)
    )

    (:action toggle_off
        :parameters (?obj - object ?agent - agent)
        :precondition (and (in_reach_of_agent ?obj)
                    (not
                        (and
                            (exists (?other_obj - object) (holds_lh
                                ?other_obj))
                            (exists (?other_obj - object) (holds_rh
                                ?other_obj))
                        )
                    )
                    (toggled_on ?obj)
        )
        :effect (not (toggled_on ?obj))
    )

    (:action left_place_nextto
        :parameters (?obj_in_hand - object ?obj - object ?agent -
            agent)
        :precondition (and (not (holds_rh ?obj))
                    (holds_lh ?obj_in_hand)
                    (in_reach_of_agent ?obj))
        :effect (and (nextto ?obj_in_hand ?obj)
                (nextto ?obj ?obj_in_hand)
                (not (holds_lh ?obj_in_hand))
                (forall (?other_obj - object) (not (holds_lh ?
                    other_obj)))
        )
    )

    (:action right_place_nextto
        :parameters (?obj_in_hand - object ?obj - object ?agent -
            agent)
        :precondition (and (not (holds_lh ?obj))
                    (holds_rh ?obj_in_hand)
                    (in_reach_of_agent ?obj))
        :effect (and (nextto ?obj_in_hand ?obj)
                (nextto ?obj ?obj_in_hand)
                (not (holds_rh ?obj_in_hand))
                (forall (?other_obj - object) (not (holds_rh ?
                    other_obj)))
        )
    )

    (:action left_transfer_contents_inside
        :parameters (?content - object ?container - object ?obj -
            object ?agent - agent)
        :precondition (and (not (holds_rh ?obj))
                    (inside ?content ?container)
                    (holds_lh ?container)
```

```
                    (in_reach_of_agent ?obj)
      )
      :effect (and (inside ?content ?obj)
                   (not (inside ?content ?container))
      )
   )

   (:action right_transfer_contents_inside
      :parameters (?content - object ?container - object ?obj -
          object ?agent - agent)
      :precondition (and (not (holds_lh ?obj))
                    (inside ?content ?container)
                    (holds_rh ?container)
                    (in_reach_of_agent ?obj)
      )
      :effect (and (inside ?content ?obj)
                   (not (inside ?content ?container))
      )
   )

   (:action left_place_nextto_ontop
      :parameters (?obj_in_hand - object ?obj1 - object ?obj2 -
          object ?agent - agent)
      :precondition (and (not (holds_rh ?obj1))
                    (not (holds_rh ?obj2))
                    (not (holds_lh ?obj1))
                    (not (holds_lh ?obj2))
                    (holds_lh ?obj_in_hand)
                    (in_reach_of_agent ?obj1)
                    (in_reach_of_agent ?obj2)
                    (not (same_obj ?obj1 ?obj2))
      )
      :effect (and (nextto ?obj_in_hand ?obj1)
                   (nextto ?obj1 ?obj_in_hand)
                   (ontop ?obj_in_hand ?obj2)
                   (not (holds_lh ?obj_in_hand))
                   (forall (?other_obj - object) (not (holds_lh ?
                       other_obj))))
   )

   (:action right_place_nextto_ontop
      :parameters (?obj_in_hand - object ?obj1 - object ?obj2 -
          object ?agent - agent)
      :precondition (and (not (holds_lh ?obj))
                    (not (holds_lh ?obj2))
                    (not (holds_rh ?obj1))
                    (not (holds_rh ?obj2))
                    (holds_rh ?obj_in_hand)
                    (in_reach_of_agent ?obj1)
                    (in_reach_of_agent ?obj2)
                    (not (same_obj ?obj1 ?obj2))
      )
      :effect (and (nextto ?obj_in_hand ?obj1)
                   (nextto ?obj1 ?obj_in_hand)
                   (ontop ?obj_in_hand ?obj2)
                   (not (holds_rh ?obj_in_hand))
                   (forall (?other_obj - object) (not (holds_rh ?
```

```
                    other_obj))))
   )

   (:action left_place_under ; place object 1 under object 2
      :parameters (?obj_in_hand - object ?obj - object ?agent -
          agent)
      :precondition (and (not (holds_rh ?obj))
                     (holds_lh ?obj_in_hand)
                     (in_reach_of_agent ?obj))
      :effect (and (under ?obj_in_hand ?obj)
                  (not (holds_lh ?obj_in_hand))
                  (forall (?other_obj - object) (not (holds_lh ?
                      other_obj))))
   )

   (:action right_place_under ; place object 1 under object 2
      :parameters (?obj_in_hand - object ?obj - object ?agent -
          agent)
      :precondition (and (not (holds_lh ?obj))
                     (holds_rh ?obj_in_hand)
                     (in_reach_of_agent ?obj))
      :effect (and (under ?obj_in_hand ?obj)
                  (not (holds_rh ?obj_in_hand))
                  (forall (?other_obj - object) (not (holds_rh ?
                      other_obj))))
   )

   (:action left_place_onfloor
      :parameters (?obj_in_hand - object ?floor - floor_n_01 ?agent
          - agent)
      :precondition (and (holds_lh ?obj_in_hand)
                     (in_reach_of_agent ?floor))
      :effect (and (onfloor ?obj_in_hand ?floor)
                  (not (holds_lh ?obj_in_hand))
                  (forall (?other_obj - object) (not (holds_lh ?
                      other_obj)))
            )
   )

   (:action right_place_onfloor
      :parameters (?obj_in_hand - object ?floor - floor_n_01 ?agent
          - agent)
      :precondition (and (holds_rh ?obj_in_hand)
                     (in_reach_of_agent ?floor))
      :effect (and (onfloor ?obj_in_hand ?floor)
                  (not (holds_rh ?obj_in_hand))
                  (forall (?other_obj - object) (not (holds_rh ?
                      other_obj)))
            )
   )
)
---

You will be presented with a PDDL problem and further instruction
    below. Your task is to:

1. Analyze the modified PDDL domain and problem files, identify
```

```
    logical errors that would make a problem unsolvable, and provide
     the corrected versions of the domain and problem files, with
    action goal modifications as needed. When fixing logical
    incompleteness, assume a reasonable default starting state (e.g
    ., objects are ‘off‘, ‘closed‘, and ‘plugged_out‘ unless
    specified otherwise).
2. Solve the PDDL problem with a sequence of actions (a plan) from
    the corrected domain and problem files. The action sequence is a
     JSON list of action strings. The actions will be executed
    sequentially in the listed order, so your sequence needs to
    follow temporal logic. Your output action plan cannot be empty
    but needs at least one action in the JSON list.

# IMPORTANT RULES

1. Do not add new action names to the domain file. You might modify
     the body of actions if utterly necessary, but no action name
    change.
2. The INPUT and OUTPUT format are detailed below. Strictly follow
    the OUTPUT FORMAT in your output, which is a JSON with three
    fields: "corrected_domain" "corrected_problem" "action_plan". If
     the domain doesn’t require modification, you can leave the "
    corrected_domain" with and empty string. If the problem doesn’t
    require modification, you can leave the "corrected_problem" with
     and empty string. Make sure the output is JSON parsable.

## INPUT FORMAT
[PDDL Problem File]
<pddl problem file>

## OUTPUT FORMAT
{{
"corrected_domain": "<corrected_pddl_domain>",
"corrected_problem": "<corrected_pddl_problem>",
"action_plan": ["<plan_of_actions_to_solve_the_problem>"]
}}

# Example 1:
Below we provide an example for your better understanding

INPUT:
[PDDL Problem File]
(define (problem bringing_in_wood)
   (:domain igibson)
   (:objects agent_n_01_1 - agent floor_n_01_1 floor_n_01_2 -
       floor_n_01 plywood_n_01_1 plywood_n_01_2 plywood_n_01_3 -
       plywood_n_01)
   (:init (onfloor plywood_n_01_1 floor_n_01_1) (onfloor
       plywood_n_01_2 floor_n_01_1) (onfloor plywood_n_01_3
       floor_n_01_1) (same_obj floor_n_01_1 floor_n_01_1) (same_obj
       floor_n_01_2 floor_n_01_2) (same_obj plywood_n_01_1
       plywood_n_01_1) (same_obj plywood_n_01_2 plywood_n_01_2) (
       same_obj plywood_n_01_3 plywood_n_01_3))
   (:goal (and (onfloor plywood_n_01_1 floor_n_01_2) (onfloor
       plywood_n_01_2 floor_n_01_2) (onfloor plywood_n_01_3
       floor_n_01_2)))
)
```

```
---
OUTPUT:
{{
"corrected_domain": "",
"corrected_problem": "",
"action_plan": [\"(navigate_to plywood_n_01_1 agent_n_01_1)\", \"(
    left_grasp plywood_n_01_1 agent_n_01_1)\", \"(navigate_to
    floor_n_01_2 agent_n_01_1)\", \"(left_place_onfloor
    plywood_n_01_1 floor_n_01_2 agent_n_01_1)\", \"(navigate_to
    plywood_n_01_2 agent_n_01_1)\", \"(left_grasp plywood_n_01_2
    agent_n_01_1)\", \"(navigate_to floor_n_01_2 agent_n_01_1)\",
    \"(left_place_onfloor plywood_n_01_2 floor_n_01_2 agent_n_01_1)
    \", \"(navigate_to plywood_n_01_3 agent_n_01_1)\", \"(left_grasp
     plywood_n_01_3 agent_n_01_1)\", \"(navigate_to floor_n_01_2
    agent_n_01_1)\", \"(left_place_onfloor plywood_n_01_3
    floor_n_01_2 agent_n_01_1)\"]
}}

# Example 2:
Below we provide another example for your better understanding

INPUT:
[PDDL Problem File]
(define (problem cleaning_floor_0)
   (:domain igibson)

   (:objects
      floor_n_01_1 - floor_n_01
      floor_n_01_2 - floor_n_01
      rag_n_01_1 - rag_n_01
      sink_n_01_1 - sink_n_01
      table_n_02_1 - table_n_02
      agent_n_01_1 - agent
   )

   (:init
      (dusty floor_n_01_1)
      (stained floor_n_01_2)
      (ontop rag_n_01_1 table_n_02_1)
   )

   (:goal
      (and
      (not (dusty floor_n_01_1))
      (not (stained floor_n_01_2))
      )
   )
)
---
OUTPUT:
{{
"corrected_domain": "",
"corrected_problem": "(define (problem cleaning_floor_0)\n (:domain
     igibson)\n\n (:objects\n \tfloor_n_01_1 - floor_n_01\n
    floor_n_01_2 - floor_n_01\n \trag_n_01_1 - rag_n_01\n \
    tsink_n_01_1 - sink_n_01\n \ttable_n_02_1 - table_n_02\n \
    tagent_n_01_1 - agent\n )\n \n (:init \n (dusty floor_n_01_1) \n
```

```
     (stained floor_n_01_2) \n (ontop rag_n_01_1 table_n_02_1) \n (
    same_obj floor_n_01_1 floor_n_01_1)\n (same_obj floor_n_01_2
    floor_n_01_2)\n (same_obj sink_n_01_1 sink_n_01_1)\n (same_obj
    rag_n_01_1 rag_n_01_1)\n (same_obj table_n_02_1 table_n_02_1)\n
    )\n \n (:goal \n (and \n (not (dusty floor_n_01_1)) \n (not (
    stained floor_n_01_2))\n )\n )\n)",
"action_plan": [\"(navigate_to sink_n_01_1 agent_n_01_1)\", \"(
    open_skip sink_n_01_1 agent_n_01_1)\", \"(toggle_on sink_n_01_1
    agent_n_01_1)\", \"(navigate_to rag_n_01_1 agent_n_01_1)\", \"(
    right_grasp rag_n_01_1 agent_n_01_1)\", \"(navigate_to
    floor_n_01_1 agent_n_01_1)\", \"(clean_dust floor_n_01_1
    agent_n_01_1)\", \"(navigate_to sink_n_01_1 agent_n_01_1)\", \"(
    right_place_inside rag_n_01_1 sink_n_01_1 agent_n_01_1)\", \"(
    soak rag_n_01_1 agent_n_01_1)\", \"(navigate_to rag_n_01_1
    agent_n_01_1)\", \"(right_grasp rag_n_01_1 agent_n_01_1)\", \"(
    navigate_to floor_n_01_2 agent_n_01_1)\", \"(
    clean_dust_and_stain floor_n_01_2 agent_n_01_1)\"]
}}

## Explanation (not part of the output)
The corrected problem added five identity (same_obj) predicates.
    Without these, the planner might wrongly use actions such as
    grasping floors for soaking in the sink, which is not possible,
    or open the sink instead of open_skip the sink, since the sink
    does not have a lid or cover and do not need to be opened.

Now, it is your turn to correct and solve the following PDDL
    problem. Comprehend and correct the domain and the problem. Then
    , solve the problem with an action plan. Remember the action
    plan needs at least one action. No explanation is needed, and
    the output marker is provided. Strictly adhere to the OUTPUT
    FORMAT and generate NOTHING ELSE.

INPUT:
[PDDL Problem File]
{problem_file}

---
OUTPUT:
```

D.3.2 VirtualHome

Template 17: VirtualHome – Action Sequencing Original Template

```
The task is to guide the robot to take actions from the current
    state to fulfill some node goals, edge goals, and action goals.
    The input will be the related objects in the scene, nodes and
    edges in the current environment, and the desired node goals,
    edge goals, and action goals. The output should be action
    commands in JSON format so that after the robot executes the
    action commands sequentially, the ending environment would
    satisfy the goals.

Data format:
Objects in the scene indicates those objects involved in the action
```

```
     execution. It follows the format: <object_name> (object_id)

Nodes and edges in the current environment shows the nodes' names,
    states and properties, and edges in the environment.
Nodes follow the format: object_name, states:... , properties:...
Edges follow the format: object_name A is ... to object_name B

Node goals show the target object states in the ending environment.
     They follow the format: object_name is ... (some state)

Edge goals show the target relationships of objects in the ending
    environment. They follow the format: object_name A is ... (some
    relationship) to object_name B.

Action goals specify the necessary actions you need to include in
    your predicted action commands sequence, and the order they
    appear in action goals should also be the RELATIVE order they
    appear in your predicted action commands sequence if there are
    more than one line. Each line in action goals include one action
     or more than one actions concatenated by OR. You only need to
    include ONE of the actions concatenated by OR in the same line.
    For example, if the action goal is:
The following action(s) should be included:
GRAB
TYPE or TOUCH
OPEN
------------------------
Then your predicted action commands sequence should include GRAB,
    either TYPE or TOUCH, and OPEN. Besides, GRAB should be executed
     earlier than TYPE or TOUCH, and TYPE or TOUCH should be
    executed earlier than OPEN.
If the action goal is:
The following action(s) should be included:
There is no action requirement.
It means there is no action you have to include in output, and you
    can use any action to achieve the node and edge goals. Warning:
    No action requirement does not mean empty output. You should
    always output some actions and their arguments.

Action commands include action names and objects. Each action's
    number of objects is fixed (0, 1, or 2), and the output should
    include object names followed by their IDs:
[]: Represents 0 objects.
[object, object_id]: Represents 1 object.
[object 1, object_1_id, object 2, object_2_id]: Represents 2
    objects.
The output must be in JSON format, where:
Dictionary keys are action names.
Dictionary values are lists containing the objects (with their IDs)
     for the corresponding action.
The order of execution is determined by the order in which the key-
    value pairs appear in the JSON dictionary.

For example, If you want to first FIND the sink and then PUTBACK a
    cup into the sink, you should express it as:
{{
  "FIND": ["sink", "sink_id"],
```

```
  "PUTBACK": ["cup", "cup_id", "sink", "sink_id"]
}}

The object of action also needs to satisfied some properties
    preconditions. For example, SWITCHON's object number is 1. To
    switch on something, the object should 'HAS_SWITCH'. The rule is
     represented as SWITCHON = ("Switch on", 1, [['HAS_SWITCH']]).
    Another example is POUR. POUR's object number is 2. To pour sth
    A into sth B, A should be pourable and drinkable, and B should
    be RECIPIENT. The rule is represented as POUR = ("Pour", 2, [['
    POURABLE', 'DRINKABLE'], ['RECIPIENT']]).

Action Definitions Format:
Each action is defined as a combination of:
Action Name (String): A descriptive name for the action.
Required Number of Parameters (Integer): The count of parameters
    needed to perform the action.
Preconditions for Each Object (List of Lists of Strings):
    Conditions that must be met for each object involved in the
    action.

Supported Actions List:
CLOSE: (1, [['CAN_OPEN']]) # Change state from OPEN to CLOSED
DRINK: (1, [['DRINKABLE', 'RECIPIENT']]) # Consume a drinkable item
FIND: (1, [[]]) # Locate and approach an item
WALK: (1, [[]]) # Move towards something
GRAB: (1, [['GRABBABLE']]) # Take hold of an item that can be
    grabbed
LOOKAT: (1, [[]]) # Direct your gaze towards something
OPEN: (1, [['CAN_OPEN']]) # Open an item that can be opened
POINTAT: (1, [[]]) # Point towards something
PUTBACK: (2, [['GRABBABLE'], []]) # Place one object back onto or
    into another
PUTIN: (2, [['GRABBABLE'], ['CAN_OPEN']]) # Insert one object into
    another
RUN: (1, [[]]) # Run towards something
SIT: (1, [['SITTABLE']]) # Sit on a suitable object
STANDUP: (0, []) # Stand up from a sitting or lying position
SWITCHOFF: (1, [['HAS_SWITCH']]) # Turn off an item with a switch
SWITCHON: (1, [['HAS_SWITCH']]) # Turn on an item with a switch
TOUCH: (1, [[]]) # Physically touch something
TURNTO: (1, [[]]) # Turn your body to face something
WATCH: (1, [[]]) # Observe something attentively
WIPE: (1, [[]]) # Clean or dry something by rubbing
PUTON: (1, [['CLOTHES']]) # Dress oneself with an item of clothing
PUTOFF: (1, [['CLOTHES']]) # Remove an item of clothing
GREET: (1, [['PERSON']]) # Offer a greeting to a person
DROP: (1, [[]]) # Let go of something so it falls
READ: (1, [['READABLE']]) # Read text from an object
LIE: (1, [['LIEABLE']]) # Lay oneself down on an object
POUR: (2, [['POURABLE', 'DRINKABLE'], ['RECIPIENT']]) # Transfer a
    liquid from one container to another
PUSH: (1, [['MOVABLE']]) # Exert force on something to move it away
     from you
PULL: (1, [['MOVABLE']]) # Exert force on something to bring it
    towards you
MOVE: (1, [['MOVABLE']]) # Change the location of an object
```

```
WASH: (1, [[]]) # Clean something by immersing and agitating it in
    water
RINSE: (1, [[]]) # Remove soap from something by applying water
SCRUB: (1, [[]]) # Clean something by rubbing it hard with a brush
SQUEEZE: (1, [['CLOTHES']]) # Compress clothes to extract liquid
PLUGIN: (1, [['HAS_PLUG']]) # Connect an electrical device to a
    power source
PLUGOUT: (1, [['HAS_PLUG']]) # Disconnect an electrical device from
     a power source
CUT: (1, [['EATABLE', 'CUTABLE']]) # Cut some food
EAT: (1, [['EATABLE']]) # Eat some food
RELEASE: (1, [[]]) # Let go of something inside the current room
TYPE: (1, [['HAS_SWITCH']]) # Type on a keyboard

Notice:
1. CLOSE action is opposed to OPEN action, CLOSE sth means changing
     the object's state from OPEN to CLOSE.

2. You cannot [PUTIN] <character> <room name>. If you want robot
    INSIDE some room, please [WALK] <room name>.

3. The subject of all these actions is <character>, that is, robot
    itself. Do not include <character> as object_name. NEVER EVER
    use character as any of the object_name, that is, the argument
    of actions.

4. The action name should be upper case without white space.

5. Importantly, if you want to apply ANY action on <object_name>,
    you should NEAR it. Therefore, you should apply WALK action as [
    WALK] <object_name> to first get near to the object before you
    apply any following actions, if you have no clue you are already
     NEAR <object_name>

6. Output only object names and their IDs, not just the names.

7. Output should not be empty! Always output some actions and their
     arguments.

8. If you want to apply an action on an object, you should WALK to
    the object first.

Input:
The relevant objects in the scene are:
{object_in_scene}

The current environment state is
{cur_change}

Node goals are:
{node_goals}

Edge goals are:
{edge_goals}

Action goals are:
{action_goals}
```

```
Please output the list of action commands in json format so that
    after the robot executes the action commands sequentially, the
    ending environment would satisfy all the node goals, edge goals
    and action goals. The dictionary keys should be action names.
    The dictionary values should be a list containing the objects of
     the corresponding action. Only output the json of action
    commands in a dictionary with nothing else.

Output:
```

Template 18: VirtualHome – Action Sequencing Updated Template (v1)

```
The task is to guide the robot to take actions from the current
    state to fulfill some node goals, edge goals, and action goals.
    The input will be the related objects in the scene, nodes and
    edges in the current environment, and the desired node goals,
    edge goals, and action goals. The output should be list of
    action commands so that after the robot executes the action
    commands sequentially, the ending environment would satisfy the
    goals.

Data format:
Objects in the scene indicates those objects involved in the action
     execution. It follows the format: <object_name> (object_id)

Nodes and edges in the current environment shows the nodes' names,
    states and properties, and edges in the environment.
Nodes follow the format: object_name, states:... , properties:...
Edges follow the format: object_name A is ... to object_name B

Node goals show the target object states in the ending environment.
     They follow the format: object_name is ... (some state)

Edge goals show the target relationships of objects in the ending
    environment. They follow the format: object_name A is ... (some
    relationship) to object_name B.

Action goals specify the necessary actions you need to include in
    your predicted action commands sequence, and the order they
    appear in action goals should also be the RELATIVE order they
    appear in your predicted action commands sequence if there are
    more than one line. Each line in action goals include one action
     or more than one actions concatenated by OR or |. You only need
     to include ONE of the actions concatenated by OR in the same
    line. For example, if the action goal is:
The following action(s) should be included:
GRAB
TYPE or TOUCH
OPEN
------------------------
Then your predicted action commands sequence should include GRAB,
    either TYPE or TOUCH, and OPEN. Besides, GRAB should be executed
     earlier than TYPE or TOUCH, and TYPE or TOUCH should be
    executed earlier than OPEN.
```

```
If the action goal is:
The following action(s) should be included:
There is no action requirement.
It means there is no action you have to include in output, and you
    can use any action to achieve the node and edge goals. Warning:
    No action requirement does not mean empty output. You should
    always output some actions and their arguments.

Action commands include action names and objects. Each action's
    number of objects is fixed (0, 1, or 2), and the output should
    include object names followed by their IDs:
[]: Represents 0 objects.
[object, object_id]: Represents 1 object.
[object 1, object_1_id, object 2, object_2_id]: Represents 2
    objects.
The output must be in a special format: a comma-separated list of
    key-value pairs that opens and closes with curly braces of form:
     {{key1: value1, key2: value2, ...}}
The keys are action names.
The values are square-bracketed lists containing the list of
    objects (followed by their IDs) as parameters for the
    corresponding action (detailed below).
The order of execution is determined by the order in which the key-
    value pairs appear.

For example, If you want to first FIND the sink and then PUTBACK a
    cup into the sink, you should express it as:
{{

  "FIND": ["sink", "sink_id"],
  "PUTBACK": ["cup", "cup_id", "sink", "sink_id"]
}}

Action Definitions Format:
<Action Name>: (<Number of Parameters>, [[<ADMISSIBLE OBJECT TYPE 1
     FOR FIRST PARAMETER>], [<ADMISSIBLE OBJECT TYPE 2 FOR FIRST
    PARAMETER>, ...], <ANOTHER LIST OF ADMISSIBLE OBJECT TYPES FOR
    SECOND PARAMETER>, ...)
Each action is defined as a combination of:
Action Name (String): A descriptive name for the action.
Required Number of Parameters (Integer): The count of parameters
    needed to perform the action.
Preconditions for Each Object (List of Lists of Strings):
    Conditions that must be met for each object involved in the
    action.

The object of action needs to satisfied some properties
    preconditions (being of some types). For example, SWITCHON's
    object number is 1. To switch on something, the object should '
    HAS_SWITCH'. The rule is represented as SWITCHON: (1, [['
    HAS_SWITCH']]). Another example is POUR. POUR's number of
    parameters is 2. To pour sth A into sth B, A should be pourable
    and drinkable, and B should be RECIPIENT. The rule is
    represented as POUR: (2, [['POURABLE', 'DRINKABLE'], ['RECIPIENT
    ']]).
```

```
Supported Actions List:
CLOSE: (1, [['CAN_OPEN']]) # Change state from OPEN to CLOSED
DRINK: (1, [['DRINKABLE', 'RECIPIENT']]) # Consume a drinkable item
FIND: (1, [[]]) # Locate and approach an item
WALK: (1, [[]]) # Move towards something
GRAB: (1, [['GRABBABLE']]) # Take hold of an item that can be
    grabbed
LOOKAT: (1, [[]]) # Direct your gaze towards something
OPEN: (1, [['CAN_OPEN']]) # Open an item that can be opened
POINTAT: (1, [[]]) # Point towards something
PUTBACK: (2, [['GRABBABLE'], []]) # Place one object back onto or
    into another
PUTIN: (2, [['GRABBABLE'], ['CAN_OPEN']]) # Insert one object into
    another
RUN: (1, [[]]) # Run towards something
SIT: (1, [['SITTABLE']]) # Sit on a suitable object
STANDUP: (0, []) # Stand up from a sitting or lying position
SWITCHOFF: (1, [['HAS_SWITCH']]) # Turn off an item with a switch
SWITCHON: (1, [['HAS_SWITCH']]) # Turn on an item with a switch
TOUCH: (1, [[]]) # Physically touch something
TURNTO: (1, [[]]) # Turn your body to face something
WATCH: (1, [[]]) # Observe something attentively
WIPE: (1, [[]]) # Clean or dry something by rubbing
PUTON: (1, [['CLOTHES']]) # Dress oneself with an item of clothing
PUTOFF: (1, [['CLOTHES']]) # Remove an item of clothing
GREET: (1, [['PERSON']]) # Offer a greeting to a person
DROP: (1, [[]]) # Let go of something so it falls
READ: (1, [['READABLE']]) # Read text from an object
LIE: (1, [['LIEABLE']]) # Lay oneself down on an object
POUR: (2, [['POURABLE', 'DRINKABLE'], ['RECIPIENT']]) # Transfer a
    liquid from one container to another
PUSH: (1, [['MOVABLE']]) # Exert force on something to move it away
     from you
PULL: (1, [['MOVABLE']]) # Exert force on something to bring it
    towards you
MOVE: (1, [['MOVABLE']]) # Change the location of an object
WASH: (1, [[]]) # Clean something by immersing and agitating it in
    water
RINSE: (1, [[]]) # Remove soap from something by applying water
SCRUB: (1, [[]]) # Clean something by rubbing it hard with a brush
SQUEEZE: (1, [['CLOTHES']]) # Compress clothes to extract liquid
PLUGIN: (1, [['HAS_PLUG']]) # Connect an electrical device to a
    power source
PLUGOUT: (1, [['HAS_PLUG']]) # Disconnect an electrical device from
     a power source
CUT: (1, [['EATABLE', 'CUTABLE']]) # Cut some food
EAT: (1, [['EATABLE']]) # Eat some food
RELEASE: (1, [[]]) # Let go of something inside the current room
TYPE: (1, [['HAS_SWITCH']]) # Type on a keyboard

Notice:
1. CLOSE action is opposed to OPEN action, CLOSE sth means changing
     the object's state from OPEN to CLOSE.

2. You cannot [PUTIN] <character> <room name>. If you want robot
    INSIDE some room, please [WALK] <room name>.
```

```
3. The subject of all these actions is <character>, that is, robot
    itself. Do not include <character> as object_name. NEVER EVER
    use character as any of the object_name, that is, the argument
    of actions.

4. The action name should be upper case without white space.

5. Importantly, if you want to apply ANY action on <object_name>,
    you should NEAR it. Therefore, you should apply WALK action as [
    WALK] <object_name> to first get near to the object before you
    apply any following actions, if you have no clue you are already
     NEAR <object_name>

6. Output only object names and their IDs, not just the names.

7. Output should not be empty! Always output some actions and their
     arguments.

8. If you want to apply an action on an object, you should WALK to
    the object first.

Example:
Below is an example for your better understanding

Input:
The relevant objects in the scene are:
Objects in the scene:

"couch, id: 352, properties: ['SURFACES', 'LIEABLE', 'SITTABLE', '
    MOVABLE']
character, id: 65, properties: []
remote_control, id: 1000, properties: ['HAS_SWITCH', 'GRABBABLE', '
    MOVABLE']
dining_room, id: 201, properties: []
television, id: 410, properties: ['HAS_PLUG', 'HAS_SWITCH', '
    LOOKABLE']
home_office, id: 319, properties: []
-----------------


The current environment state is
Nodes:
television, states: ['CLEAN', 'OFF', 'PLUGGED_IN'], properties:['
    HAS_PLUG', 'HAS_SWITCH', 'LOOKABLE']


Edges:
<couch> (352) is NEAR to <remote_control> (1000)
<character> (65) is INSIDE to <dining_room> (201)
<remote_control> (1000) is NEAR to <couch> (352)
-----------------


Node goals are:
television is ON
television is PLUGGED_IN
-----------------
```

```
Edge goals are:
character is FACING to television
-----------------


Action goals are:
The following action(s) should be included:
LOOKAT or WATCH
LOOKAT|WATCH
-----------------


Output:
{{\n\"TURNTO\": [\"television\", \"410\"],\n\"WALK\": [\"television
    \", \"410\"],\n\"SWITCHON\": [\"television\", \"410\"],\n\"
    LOOKAT\": [\"television\", \"410\"]\n}}


Now is your turn. Please output the list of action commands in the
    detailed curly-bracketed list format so that after the robot
    executes the action commands sequentially, the ending
    environment would satisfy all the node goals, edge goals and
    action goals. The keys should be action names. The dictionary
    values should be a list containing the object parameters of the
    corresponding action. No explanation, prefix, or suffix is
    needed. Only output the curly-bracketed list and nothing else.

Input:
The relevant objects in the scene are:
{object_in_scene}

The current environment state is
{cur_change}

Node goals are:
{node_goals}

Edge goals are:
{edge_goals}

Action goals are:
{action_goals}

Output:
```

Template 19: VirtualHome – Action Sequencing PDDL Template (v2)

```
# Background
You are an expert PDDL (Planning Domain Definition Language)
    analyzer and debugger. You are given the following PDDL domain
    file:

---
```

```
[PDDL Domain File]
(define (domain virtualhome)
   ; (:requirements :typing)
   (:requirements :strips :adl :typing :negative-preconditions)
   (:types
      object character ; Define 'object' and 'character' as types
   )

   (:predicates
      (closed ?obj - object) ; obj is closed
      (open ?obj - object) ; obj is open
      (on ?obj - object) ; obj is turned on, or it is activated
      (off ?obj - object) ; obj is turned off, or it is deactivated
      (sitting ?char - character) ; char is sitting, and this
         represents a state of a character
      (dirty ?obj - object) ; obj is dirty
      (clean ?obj - object) ; obj is clean
      (lying ?char - character) ; char is lying
      (plugged_in ?obj - object) ; obj is plugged in
      (plugged_out ?obj - object) ; obj is unplugged
      (obj_ontop ?obj1 ?obj2 - object) ; obj1 is on top of obj2
      (ontop ?char - character ?obj - object) ; char is on obj
      (on_char ?obj - object ?char - character) ; obj is on char
      (inside ?char - character ?obj - object) ; char is inside obj
      (obj_inside ?obj1 ?obj2 - object) ; obj1 is inside obj2
      (between ?obj1 ?obj2 ?obj3 - object) ; obj1 is between obj2
         and obj3
      (obj_next_to ?obj1 ?obj2 - object) ; obj1 is close to or next
          to obj2
      (next_to ?char - character ?obj - object) ; char is close to
         or next to obj
      (facing ?char - character ?obj - object) ; char is facing obj
      (holds_rh ?char - character ?obj - object) ; char is holding
         obj with right hand
      (holds_lh ?char - character ?obj - object) ; char is holding
         obj with left hand
      (grabbable ?obj - object) ; obj can be grabbed
      (cuttable ?obj - object) ; obj can be cut
      (can_open ?obj - object) ; obj can be opened
      (readable ?obj - object) ; obj can be read
      (has_paper ?obj - object) ; obj has paper
      (movable ?obj - object) ; obj is movable
      (pourable ?obj - object) ; obj can be poured from
      (cream ?obj - object) ; obj is cream
      (has_switch ?obj - object) ; obj has a switch
      (lookable ?obj - object) ; obj can be looked at
      (has_plug ?obj - object) ; obj has a plug
      (drinkable ?obj - object) ; obj is drinkable
      (body_part ?obj - object) ; obj is a body part
      (recipient ?obj - object) ; obj is a recipient
      (containers ?obj - object) ; obj is a container
      (cover_object ?obj - object) ; obj is a cover object
      (surfaces ?obj - object) ; obj has surfaces
      (sittable ?obj - object) ; obj can be sat on
      (lieable ?obj - object) ; obj can be lied on
      (person ?obj - object) ; obj is a person
      (hangable ?obj - object) ; obj can be hanged
```

```
   (clothes ?obj - object) ; obj is clothes
   (eatable ?obj - object) ; obj is eatable
   (found ?obj - object) ; obj is found
   )

(:action close
   :parameters (?char - character ?obj - object)
   :precondition (and
             (can_open ?obj)
             (open ?obj)
             (next_to ?char ?obj)
   )
   :effect (and
         (closed ?obj)
         (not (open ?obj))
   )
)

(:action drink
   :parameters (?char - character ?obj - object)
   :precondition (or
             (and
             (drinkable ?obj)
             (holds_lh ?char ?obj)
             )
             (and
             (drinkable ?obj)
             (holds_rh ?char ?obj)
             )
             (and
             (recipient ?obj)
             (holds_lh ?char ?obj)
             )
             (and
             (recipient ?obj)
             (holds_rh ?char ?obj)
             )
   )
   :effect (and)
)

(:action find
   :parameters (?char - character ?obj - object)
   :precondition (not(next_to ?char ?obj))
   :effect (found ?obj)
)

(:action walk
   :parameters (?char - character ?obj - object)
   :precondition (and
             (not (sitting ?char))
             (not (lying ?char))
             (not (next_to ?char ?obj))
   )
   :effect (and
         (next_to ?char ?obj)
         (forall (?far_obj - object)
```

```
            (when (not (obj_next_to ?far_obj ?obj)) (not (
                next_to ?char ?far_obj)))
         )
         (forall (?close_obj - object)
            (when (obj_next_to ?close_obj ?obj) (next_to ?char ?
                close_obj))
         )
         (forall (?inside_obj - object)
            (when (obj_inside ?inside_obj ?obj) (next_to ?char ?
                inside_obj))
         )
         (forall (?ontop_obj - object)
            (when (obj_ontop ?ontop_obj ?obj) (next_to ?char ?
                ontop_obj))
         )
         (forall (?under_obj - object)
            (when (obj_ontop ?obj ?under_obj) (next_to ?char ?
                under_obj))
         )
   )
)

(:action grab
   :parameters (?char - character ?obj - object)
   :precondition (and
             (grabbable ?obj)
             (next_to ?char ?obj)
             (not (exists (?obj2 - object) (and (obj_inside ?obj
                  ?obj2) (or (closed ?obj2) (on ?obj2)))))
             (not (and (exists (?obj3 - object) (holds_lh ?char
                 ?obj3)) (exists (?obj4 - object) (holds_rh ?
                 char ?obj4))))
             )
   :effect (and
         (when (exists (?obj3 - object) (holds_lh ?char ?obj3))
             (holds_rh ?char ?obj))
         (when (exists (?obj4 - object) (holds_rh ?char ?obj4))
             (holds_lh ?char ?obj))
         (when (and
               (not (exists (?obj3 - object) (holds_lh ?char ?
                   obj3)))
               (not (exists (?obj4 - object) (holds_rh ?char ?
                   obj4)))
            )
            (holds_rh ?char ?obj) ; Default to right hand
         )
         (forall (?obj2 - object)
            (when (obj_inside ?obj ?obj2)
               (not (obj_inside ?obj ?obj2))
            )
         )
   )
)

(:action lookat
   :parameters (?char - character ?obj - object)
   :precondition (and
```

```
                (lookable ?obj)
                (facing ?char ?obj)
      )
      :effect (and)
   )

   (:action open
      :parameters (?char - character ?obj - object)
      :precondition (and
                (can_open ?obj)
                (closed ?obj)
                (next_to ?char ?obj)
                (not (on ?obj))
      )
      :effect (and
            (open ?obj)
            (not (closed ?obj))
      )
   )

   (:action putin
      :parameters (?char - character ?obj1 - object ?obj2 - object)
      :precondition (or
                 (and
                 (next_to ?char ?obj2)
                 (holds_lh ?char ?obj1)
                 (not (can_open ?obj2))
                 )
                 (and
                 (next_to ?char ?obj2)
                 (holds_lh ?char ?obj1)
                 (open ?obj2)
                 )
                 (and
                 (next_to ?char ?obj2)
                 (holds_rh ?char ?obj1)
                 (not (can_open ?obj2))
                 )
                 (and
                 (next_to ?char ?obj2)
                 (holds_rh ?char ?obj1)
                 (open ?obj2)
                 )
      )
      :effect (and
            (obj_inside ?obj1 ?obj2)
            (not (holds_lh ?char ?obj1))
            (not (holds_rh ?char ?obj1))
      )
   )

   (:action sit
      :parameters (?char - character ?obj - object)
      :precondition (and
                (next_to ?char ?obj)
                (sittable ?obj)
                (not (sitting ?char))
```

```
   )
   :effect (and
         (sitting ?char)
         (ontop ?char ?obj)
   )
)

(:action standup
   :parameters (?char - character)
   :precondition (or
            (sitting ?char)
            (lying ?char)
   )
   :effect (and
         (not (sitting ?char))
         (not (lying ?char))
         (not (ontop ?char ?obj))
   )
)


(:action switchoff
   :parameters (?char - character ?obj - object)
   :precondition (and
            (has_switch ?obj)
            (on ?obj)
            (next_to ?char ?obj)
   )
   :effect (and
         (off ?obj)
         (not (on ?obj))
   )
)

(:action switchon
   :parameters (?char - character ?obj - object)
   :precondition (and
            (has_switch ?obj)
            (off ?obj)
            (plugged_in ?obj)
            (next_to ?char ?obj)

   )
   :effect (and
         (on ?obj)
         (not (off ?obj))
   )
)

(:action touch
   :parameters (?char - character ?obj - object)
   :precondition (or
            (and
            (readable ?obj)
            (holds_lh ?char ?obj)
            (not (exists (?obj2 - object) (and (obj_inside ?obj
                 ?obj2) (closed ?obj2))))
```

```
                )
                (and
                (readable ?obj)
                (holds_rh ?char ?obj)
                (not (exists (?obj2 - object) (and (obj_inside ?obj
                    ?obj2) (closed ?obj2))))
                )
      )
      :effect (and)
   )

   (:action turnto
      :parameters (?char - character ?obj - object)
      :precondition (and
                  (not (exists (?obj2 - object) (and (obj_inside ?
                      obj ?obj2) (closed ?obj2))))
      )
      :effect (and
               (forall (?obj2 - object)
                  (when (not (and (obj_inside ?obj ?obj2) (open ?
                      obj2)))
                     (not (facing ?char ?obj2))
                  )
               )
               (facing ?char ?obj)
      )
   )

   (:action watch
      :parameters (?char - character ?obj - object)
      :precondition (and
                (lookable ?obj)
                (facing ?char ?obj)
                (not (exists (?obj2 - object) (and (obj_inside ?obj
                    ?obj2) (closed ?obj2))))
      )
      :effect (and)
   )

   (:action wipe
      :parameters (?char - character ?obj1 - object ?obj2 - object)
      :precondition (or
                (and
                (next_to ?char ?obj1)
                (holds_lh ?char ?obj2)
                )
                (and
                (next_to ?char ?obj1)
                (holds_rh ?char ?obj2)
                )
      )
      :effect (and
            (clean ?obj1)
            (not (dirty ?obj1))
      )
   )
```

```
    (:action obj_puton
       :parameters (?char - character ?obj1 - object ?obj2 - object)
       :precondition (or
                   (and
                   (next_to ?char ?obj2)
                   (holds_lh ?char ?obj1)
                   )
                   (and
                   (next_to ?char ?obj2)
                   (holds_rh ?char ?obj1)
                   )
       )
       :effect (and
             (obj_next_to ?obj1 ?obj2)
             (obj_ontop ?obj1 ?obj2)
             (when (holds_lh ?char ?obj1) (not (holds_lh ?char ?obj1)
                ))
             (when (holds_rh ?char ?obj1) (not (holds_rh ?char ?obj1)
                ))
       )
    )
    (:action puton
       :parameters (?char - character ?obj - object)
       :precondition (or
                   (holds_lh ?char ?obj)
                   (holds_rh ?char ?obj)
                )
       :effect (and
             (on_char ?obj ?char)
             (when (holds_lh ?char ?obj1) (not (holds_lh ?char ?obj1)
                ))
             (when (holds_rh ?char ?obj1) (not (holds_rh ?char ?obj1)
                ))
       )
    )

    (:action putoff_lh
       :parameters (?char - character ?obj - object)
       :precondition (and
                   (on_char ?obj ?char)
                   (not (holds_lh ?char ?obj))
       )
       :effect (and
             (not (on_char ?obj ?char))
             (holds_lh ?char ?obj)
       )
    )

    (:action putoff_rh
       :parameters (?char - character ?obj - object)
       :precondition (and
                   (on_char ?obj ?char)
                   (not (holds_rh ?char ?obj))
       )
       :effect (and
             (not (on_char ?obj ?char))
             (holds_rh ?char ?obj)
```

```
      )
   )

   (:action drop
      :parameters (?char - character ?obj - object ?room - object)
      :precondition (or
                 (and
                 (holds_lh ?char ?obj)
                 (obj_inside ?obj ?room)
                 )
                 (and
                 (holds_rh ?char ?obj)
                 (obj_inside ?obj ?room)
                 )
      )
      :effect (and
            (not (holds_lh ?char ?obj))
            (not (holds_rh ?char ?obj))
      )
   )

   (:action read
      :parameters (?char - character ?obj - object)
      :precondition (or
                 (and
                 (readable ?obj)
                 (holds_lh ?char ?obj)
                 )
                 (and
                 (readable ?obj)
                 (holds_rh ?char ?obj)
                 )
      )
      :effect (and)
   )

   (:action lie
      :parameters (?char - character ?obj - object)
      :precondition (and
                 (lieable ?obj)
                 (next_to ?char ?obj)
                 (not (lying ?char))
      )
      :effect (and
            (lying ?char)
            (ontop ?char ?obj)
            (not (sitting ?char))
      )
   )

   (:action pour
      :parameters (?char - character ?obj1 - object ?obj2 - object)
      :precondition (or
                 (and
                 (pourable ?obj1)
                 (holds_lh ?char ?obj1)
                 (recipient ?obj2)
```

```
                (next_to ?char ?obj2)
                )
                (and
                (pourable ?obj1)
                (holds_rh ?char ?obj1)
                (recipient ?obj2)
                (next_to ?char ?obj2)
                )
                (and
                (drinkable ?obj1)
                (holds_lh ?char ?obj1)
                (recipient ?obj2)
                (next_to ?char ?obj2)
                )
                (and
                (drinkable ?obj1)
                (holds_rh ?char ?obj1)
                (recipient ?obj2)
                (next_to ?char ?obj2)
                )
      )
      :effect (obj_inside ?obj1 ?obj2)
   )

   (:action type
      :parameters (?char - character ?obj - object)
      :precondition (and
                (has_switch ?obj)
                (next_to ?char ?obj)
      )
      :effect (and)
   )

   (:action move
      :parameters (?char - character ?obj - object)
      :precondition (and
                (movable ?obj)
                (next_to ?char ?obj)
                (not (exists (?obj2 - object) (and (obj_inside ?obj
                     ?obj2) (closed ?obj2))))
      )
      :effect (and)
   )

   (:action wash
      :parameters (?char - character ?obj - object)
      :precondition (and
               (next_to ?char ?obj)
               (exists (?basin - obj)
                  (and
                     (containers ?obj)
                     (obj_inside ?obj ?basin)
                  )
               )
      )
      :effect (and
            (clean ?obj)
```

```
            (not (dirty ?obj))
      )
   )

   (:action squeeze
      :parameters (?char - character ?obj - object)
      :precondition (and
                (next_to ?char ?obj)
                (clothes ?obj)
      )
      :effect (and)
   )

   (:action plugin
      :parameters (?char - character ?obj - object)
      :precondition (or
                  (and
                  (next_to ?char ?obj)
                  (has_plug ?obj)
                  (plugged_out ?obj)
                  )
                  (and
                  (next_to ?char ?obj)
                  (has_switch ?obj)
                  (plugged_out ?obj)
                  )
                )
      :effect (and
            (plugged_in ?obj)
            (not (plugged_out ?obj))
      )
   )

   (:action plugout
      :parameters (?char - character ?obj - object)
      :precondition (and
                (next_to ?char ?obj)
                (has_plug ?obj)
                (plugged_in ?obj)
                (not (on ?obj))
      )
      :effect (and
            (plugged_out ?obj)
            (not (plugged_in ?obj))
      )
   )

   (:action cut
      :parameters (?char - character ?obj - object)
      :precondition (and
                (next_to ?char ?obj)
                (eatable ?obj)
                (cuttable ?obj)
      )
      :effect (and)
   )
```

```
   (:action eat
      :parameters (?char - character ?obj - object)
      :precondition (and
                (next_to ?char ?obj)
                (eatable ?obj)
      )
      :effect (and)
   )

   (:action sleep
      :parameters (?char - character)
      :precondition (or
                (lying ?char)
                (sitting ?char)
      )
      :effect (and)
   )

   (:action wake_up
      :parameters (?char - character)
      :precondition (or
                (lying ?char)
                (sitting ?char)
      )
      :effect (and)
   )
)
---

You will be presented with a PDDL problem and further instruction
    below. Your task is to:

1. Modify the PDDL domain to accomodate action goals if a problem
    comes with action goal requirements. For each required actions
    analyze the natural language command to identify the action
    predicate with correct parameters that needs to be performed,
    then add to the domain a special predicate that marks the
    satisfaction of that action goal. You might need to modify the
    effect of the action to trigger this special predicate. There
    could be multiple action goals. Action goals listed on the same
    line and connected with the word "or" mean only one among them
    is needed to be add to the goal. Action goals listed on
    different lines are implicitly connected with "and" so they need
     to be satisfied together. If there is no action goal
    requirement (nothing in section [Action Goals]), skip this
    subtask.
2. Analyze the modified PDDL domain and problem files, identify
    logical errors that would make a problem unsolvable, and provide
     the corrected versions of the domain and problem files, with
    action goal modifications as needed. (a) When fixing logical
    incompleteness of missing intial states, assume a reasonable
    default starting state (e.g., objects are `off`, `closed`, and `
    plugged_out` unless specified otherwise). (b) When the problem
    is listed with multiple rooms and the character is in one of
    them, use common sense to infer if the objects relevant to the
    goals are likely to be in which room and make such implied
    conditions explicit `inside` predicates in the :init block. In
```

```
    many cases, such problems require the character to walk to the
    target room first.
3. Solve the PDDL problem with a sequence of actions (a plan) from
    the corrected domain and problem files. The action sequence is a
     JSON list of action strings. The actions will be executed
    sequentially in the listed order, so your sequence needs to
    follow temporal logic. Your output action plan cannot be empty
    but needs at least one action in the JSON list.

# IMPORTANT RULES

1. Do not add new action names to the domain file. You might modify
     the body of actions if utterly necessary, but no action name
    change.
2. The natural language command might not be compatible or fully
    reflect the state and action goals. It is not the ground truth
    but only a description with similar contexts to the task at hand
     and should only be used to infer the action goals. Sometimes
    action goals are required even when there is no natural language
     command available (nothing in [Natural Language Command]
    section), then you have to comprehend and conceptualize the
    action goal predicates yourself for domain and problem
    modifications.
3. Do not put new objects to the problem file. You might modify
    intial states if utterly necessary (e.g., to prevent deadlocks)
    and with reasonable assumptions, but do not add new objects.
4. Do not negate existing initial states. If some goal states are
    already satisfied from the beginning, you do not need to change
    them, less work is good.
4. The INPUT and OUTPUT format are detailed below. Strictly follow
    the OUTPUT FORMAT in your output, which is a JSON with three
    fields: "corrected_domain" "corrected_problem" "action_plan". If
     the domain doesn't require modification, you can leave the "
    corrected_domain" with and empty string. If the problem doesn't
    require modification, you can leave the "corrected_problem" with
     and empty string. Make sure the output is JSON parsable.

## INPUT FORMAT
[Natural Language Command]
<natural language command>

---
[Action Goals]
<action goals>

---
[PDDL Problem File]
<pddl problem file>

## OUTPUT FORMAT
{{
"corrected_domain": "<corrected_pddl_domain>",
"corrected_problem": "<corrected_pddl_problem>",
"action_plan": ["<plan_of_actions_to_solve_the_problem>"]
}}

# Example 1:
```

```
Below we provide an example for your better understanding

INPUT:
[Natural Language Command]
Goal name: Wash clothes

Goal description: Walk to the kitchen and find the basket of
    clothes. Put the soap and clothes into the washing machine. Turn
     on the washing machine.

---
[Action Goals]

---
[PDDL Problem File]
(define (problem scene_1_27_2)
   (:domain virtualhome)
   (:objects
         character_id65 - character
   bathroom_id1 dining_room_id201 basket_for_clothes_id1000
       washing_machine_id1001 soap_id1002 clothes_jacket_id1003 -
       object
   )
   (:init
      (can_open basket_for_clothes_id1000)
      (can_open washing_machine_id1001)
      (clothes clothes_jacket_id1003)
      (containers basket_for_clothes_id1000)
      (containers washing_machine_id1001)
      (cream soap_id1002)
      (grabbable basket_for_clothes_id1000)
      (grabbable clothes_jacket_id1003)
      (grabbable soap_id1002)
      (hangable clothes_jacket_id1003)
      (has_plug washing_machine_id1001)
      (has_switch washing_machine_id1001)
      (inside character_id65 bathroom_id1)
      (movable basket_for_clothes_id1000)
      (movable clothes_jacket_id1003)
      (movable soap_id1002)
      (obj_inside clothes_jacket_id1003 washing_machine_id1001)
      (obj_next_to basket_for_clothes_id1000 clothes_jacket_id1003)
      (obj_next_to basket_for_clothes_id1000 soap_id1002)
      (obj_next_to clothes_jacket_id1003 basket_for_clothes_id1000)
      (obj_next_to soap_id1002 basket_for_clothes_id1000)
      (recipient washing_machine_id1001)
   )
   (:goal
      (and
         (closed washing_machine_id1001)
         (obj_ontop clothes_jacket_id1003 washing_machine_id1001)
         (obj_ontop soap_id1002 washing_machine_id1001)
         (on washing_machine_id1001)
         (plugged_in washing_machine_id1001)
      )
   )
)
```

```
---
OUTPUT:
{{
"corrected_domain": "",
"corrected_problem": "(define (problem scene_1_27_2)\n (:domain
    virtualhome)\n (:objects\n character_id65 - character\n
    bathroom_id1 dining_room_id201 basket_for_clothes_id1000
    washing_machine_id1001 soap_id1002 clothes_jacket_id1003 -
    object\n )\n (:init\n (can_open basket_for_clothes_id1000)\n (
    can_open washing_machine_id1001)\n (clothes
    clothes_jacket_id1003)\n (containers basket_for_clothes_id1000)\
    n (containers washing_machine_id1001)\n (cream soap_id1002)\n (
    grabbable basket_for_clothes_id1000)\n (grabbable
    clothes_jacket_id1003)\n (grabbable soap_id1002)\n (hangable
    clothes_jacket_id1003)\n (has_plug washing_machine_id1001)\n (
    has_switch washing_machine_id1001)\n (inside character_id65
    bathroom_id1)\n (movable basket_for_clothes_id1000)\n (movable
    clothes_jacket_id1003)\n (movable soap_id1002)\n (obj_inside
    clothes_jacket_id1003 washing_machine_id1001)\n (obj_next_to
    basket_for_clothes_id1000 clothes_jacket_id1003)\n (obj_next_to
    basket_for_clothes_id1000 soap_id1002)\n (obj_next_to
    clothes_jacket_id1003 basket_for_clothes_id1000)\n (obj_next_to
    soap_id1002 basket_for_clothes_id1000)\n (recipient
    washing_machine_id1001)\n (off washing_machine_id1001) ; Specify
     it starts 'off'\n (plugged_out washing_machine_id1001) ;
    Specify it starts 'plugged_out'\n (closed washing_machine_id1001
    ) ; Specify it starts 'closed'\n )\n (:goal\n (and\n (closed
    washing_machine_id1001)\n (obj_ontop clothes_jacket_id1003
    washing_machine_id1001)\n (obj_ontop soap_id1002
    washing_machine_id1001)\n (on washing_machine_id1001)\n (
    plugged_in washing_machine_id1001)\n )\n )\n)\n",
"action_plan": [\"(walk character_id65 washing_machine_id100)\",
    \"(plugin character_id65 washing_machine_id1001)\", \"(open
    character_id65 washing_machine_id1001)\", \"(grab character_id65
     clothes_jacket_id1003)\", \"(obj_puton character_id65
    clothes_jacket_id1003 washing_machine_id1001)\", \"(close
    character_id65 washing_machine_id1001)\", \"(switchon
    character_id65 washing_machine_id1001)\", \"(walk character_id65
     basket_for_clothes_id1000)\", \"(grab character_id65
    soap_id1002)\", \"(walk character_id65 clothes_jacket_id1003)\",
     \"(obj_puton character_id65 soap_id1002 washing_machine_id1001)
    \"]
}}

## Explanation (not part of the output):
The core issue is that the washing machine's initial state is not
    fully defined. For mutually exclusive states (like on/off, open/
    closed, plugged_in/plugged_out), the (:init) block specifies
    neither state. If an object is defined in the domain to have
    these states, the '(:init)' block **must** specify one of these
    states. If neither is specified, the object is in a logical
    limbo, and any action requiring one of these states as a
    precondition will fail.

Here is a breakdown of the unreachable goals:
```

```
1. The on/off Deadlock
Goal: (on washing_machine_id1001)

Action to Achieve: (:action switchon)

Precondition for switchon: This action requires the predicate (off
    washing_machine_id1001) to be true.

The Problem: The (:init) block does not state that the washing
    machine is (off). It also doesn't state that it's (on).

The Deadlock:
You can't use switchon because (off) is not true.
You can't use switchoff (which would add the (off) state) because
    its precondition is (on), which is also not true.
Therefore, the state of the washing machine can never be changed
    from its initial, undefined "on/off" status.

2. The plugged_in/plugged_out Deadlock
Goal: (plugged_in washing_machine_id1001)

Action to Achieve: (:action plugin)

Precondition for plugin: This action requires the predicate (
    plugged_out washing_machine_id1001) to be true.

The Problem: The (:init) block does not state that the washing
    machine is (plugged_out).

The Deadlock: This is the same as the on/off problem.
You can't use plugin because (plugged_out) is not true.
You can't use plugout (to make plugged_out true) because its
    precondition is (plugged_in), which is also not true.
The machine can never be plugged in.

3. The open/closed Deadlock
Goal: (closed washing_machine_id1001)

Action to Achieve: (:action close)

Precondition for close: This action requires the predicate (open
    washing_machine_id1001) to be true.

The Problem: The (:init) block does not state that the washing
    machine is (open).

The Deadlock:
You can't use close because (open) is not true.
You can't use open (to make open true) because its precondition is
    (closed), which is also not true.
The machine can never be opened or closed.


# Example 2:
Below we provide an example for your better understanding

INPUT:
```

```
[Natural Language Command]
Goal name: Watch TV

Goal description: I go to living room, sit on sofa, find the remote
     control on the table, switch on TV and watch the TV.

---
[Action Goals]
LOOKAT or WATCH

---
[PDDL Problem File]
(define (problem scene_1_21_2)
   (:domain virtualhome)
   (:objects
         character_id65 - character
   couch_id352 table_id355 remote_control_id1000 dining_room_id201
       television_id410 home_office_id319 - object
   )
   (:init
      (clean television_id410)
      (grabbable remote_control_id1000)
      (has_plug television_id410)
      (has_switch remote_control_id1000)
      (has_switch television_id410)
      (inside_room character_id65 dining_room_id201)
      (lieable couch_id352)
      (lookable television_id410)
      (movable couch_id352)
      (movable remote_control_id1000)
      (movable table_id355)
      (obj_next_to couch_id352 remote_control_id1000)
      (obj_next_to remote_control_id1000 couch_id352)
      (off television_id410)
      (plugged_in television_id410)
      (sittable couch_id352)
      (surfaces couch_id352)
      (surfaces table_id355)
   )
   (:goal
      (and
         (facing character_id65 television_id410)
         (on television_id410)
         (plugged_in television_id410)
      )
   )
)

_____
OUTPUT:
{{
"corrected_domain": "(define (domain virtualhome)\n (:requirements :
    typing)\n (:types \n object character ; Define 'object' and '
    character' as types\n )\n\n (:predicates\n (closed ?obj - object
    ) ; obj is closed\n (open ?obj - object) ; obj is open\n (on ?
    obj - object) ; obj is turned on, or it is activated\n (off ?obj
     - object) ; obj is turned off, or it is deactivated\n (sitting
```

```
?char - character) ; char is sitting, and this represents a
state of a character\n (dirty ?obj - object) ; obj is dirty\n (
clean ?obj - object) ; obj is clean\n (lying ?char - character)
; char is lying\n (plugged_in ?obj - object) ; obj is plugged in
\n (plugged_out ?obj - object) ; obj is unplugged\n (obj_ontop ?
obj1 ?obj2 - object) ; obj1 is on top of obj2\n (ontop ?char -
character ?obj - object) ; char is on obj\n (on_char ?obj -
object ?char - character) ; obj is on char\n (inside ?char -
character ?obj - object) ; char is inside obj\n (obj_inside ?
obj1 ?obj2 - object) ; obj1 is inside obj2\n (between ?obj1 ?
obj2 ?obj3 - object) ; obj1 is between obj2 and obj3\n (
obj_next_to ?obj1 ?obj2 - object) ; obj1 is close to or next to
obj2\n (next_to ?char - character ?obj - object) ; char is close
 to or next to obj\n (facing ?char - character ?obj - object) ;
char is facing obj\n (holds_rh ?char - character ?obj - object)
; char is holding obj with right hand\n (holds_lh ?char -
character ?obj - object) ; char is holding obj with left hand\n
(grabbable ?obj - object) ; obj can be grabbed\n (cuttable ?obj
- object) ; obj can be cut\n (can_open ?obj - object) ; obj can
be opened\n (readable ?obj - object) ; obj can be read\n (
has_paper ?obj - object) ; obj has paper\n (movable ?obj -
object) ; obj is movable\n (pourable ?obj - object) ; obj can be
 poured from\n (cream ?obj - object) ; obj is cream\n (
has_switch ?obj - object) ; obj has a switch\n (lookable ?obj -
object) ; obj can be looked at\n (has_plug ?obj - object) ; obj
has a plug\n (drinkable ?obj - object) ; obj is drinkable\n (
body_part ?obj - object) ; obj is a body part\n (recipient ?obj
- object) ; obj is a recipient\n (containers ?obj - object) ;
obj is a container\n (cover_object ?obj - object) ; obj is a
cover object\n (surfaces ?obj - object) ; obj has surfaces\n (
sittable ?obj - object) ; obj can be sat on\n (lieable ?obj -
object) ; obj can be lied on\n (person ?obj - object) ; obj is a
 person\n (hangable ?obj - object) ; obj can be hanged\n (
clothes ?obj - object) ; obj is clothes\n (eatable ?obj - object
) ; obj is eatable\n (did_watch ?char - character ?obj - object)
 ; char has watched obj\n (did_lookat ?char - character ?obj -
object) ; char has looked at obj\n )\n\n (:action close\n :
parameters (?char - character ?obj - object)\n :precondition (
and\n (can_open ?obj)\n (open ?obj)\n (next_to ?char ?obj)\n )\n
 :effect (and\n (closed ?obj)\n (not (open ?obj))\n ) \n )\n\n
(:action drink\n :parameters (?char - character ?obj - object)\n
 :precondition (or\n (and\n (drinkable ?obj)\n (holds_lh ?char ?
obj)\n )\n (and\n (drinkable ?obj)\n (holds_rh ?char ?obj)\n )\n
 (and\n (recipient ?obj)\n (holds_lh ?char ?obj)\n )\n (and\n (
recipient ?obj)\n (holds_rh ?char ?obj)\n )\n ) \n :effect (and)
 \n )\n\n (:action find\n :parameters (?char - character ?obj -
object)\n :precondition (next_to ?char ?obj)\n :effect (and)\n )
\n\n (:action walk\n :parameters (?char - character ?obj -
object)\n :precondition (and\n (not (sitting ?char))\n (not (
lying ?char))\n )\n :effect (and\n (next_to ?char ?obj)\n (
forall (?far_obj - object) \n (when (not (obj_next_to ?far_obj ?
obj)) (not (next_to ?char ?far_obj)))\n )\n (forall (?close_obj
- object)\n (when (obj_next_to ?close_obj ?obj) (next_to ?char ?
close_obj))\n )\n (forall (?inside_obj - object)\n (when (
obj_inside ?inside_obj ?obj) (next_to ?char ?inside_obj))\n ) \n
 (forall (?ontop_obj - object)\n (when (obj_ontop ?ontop_obj ?
obj) (next_to ?char ?ontop_obj))\n )\n (forall (?under_obj -
```

```
object)\n (when (obj_ontop ?obj ?under_obj) (next_to ?char ?
under_obj))\n )\n )\n )\n\n (:action grab\n :parameters (?char -
 character ?obj - object)\n :precondition (and\n (grabbable ?obj
)\n (next_to ?char ?obj)\n (not (exists (?obj2 - object) (and (
obj_inside ?obj ?obj2) (or (closed ?obj2) (on ?obj2))))))\n (not
(and (exists (?obj3 - object) (holds_lh ?char ?obj3)) (exists (?
obj4 - object) (holds_rh ?char ?obj4))))\n )\n :effect (and\n (
when (exists (?obj3 - object) (holds_lh ?char ?obj3)) (holds_rh
?char ?obj))\n (when (exists (?obj4 - object) (holds_rh ?char ?
obj4)) (holds_lh ?char ?obj))\n (when \n (not (and (exists (?
obj3 - object) (holds_lh ?char ?obj3)) (exists (?obj4 - object)
(holds_rh ?char ?obj4))))\n (holds_rh ?char ?obj)\n )\n (when \n
 (not (exists (?obj2 - object) (and (obj_inside ?obj ?obj2) (
closed ?obj2)))) \n (not (obj_inside ?obj ?obj2))\n )\n ) \n )\n
\n (:action lookat\n :parameters (?char - character ?obj -
object)\n :precondition (and \n (lookable ?obj) \n (facing ?char
 ?obj) \n )\n :effect (and\n (did_lookat ?char ?obj)\n )\n )\n\n
 (:action open\n :parameters (?char - character ?obj - object)\n
 :precondition (and\n (can_open ?obj)\n (closed ?obj)\n (next_to
 ?char ?obj)\n (not (on ?obj))\n ) \n :effect (and\n (open ?obj)
\n (not (closed ?obj))\n ) \n )\n\n (:action putin\n :parameters
 (?char - character ?obj1 - object ?obj2 - object)\n :
precondition (or\n (and\n (next_to ?char ?obj2)\n (holds_lh ?
char ?obj1)\n (not (can_open ?obj2))\n )\n (and\n (next_to ?char
 ?obj2)\n (holds_lh ?char ?obj1)\n (open ?obj2)\n )\n (and\n (
next_to ?char ?obj2)\n (holds_rh ?char ?obj1)\n (not (can_open ?
obj2))\n )\n (and\n (next_to ?char ?obj2)\n (holds_rh ?char ?
obj1)\n (open ?obj2)\n )\n )\n :effect (and\n (obj_inside ?obj1
?obj2)\n (not (holds_lh ?char ?obj1))\n (not (holds_rh ?char ?
obj1))\n ) \n )\n\n (:action sit\n :parameters (?char -
character ?obj - object)\n :precondition (and\n (next_to ?char ?
obj)\n (sittable ?obj)\n (not (sitting ?char))\n )\n :effect (
and\n (sitting ?char)\n (ontop ?char ?obj)\n ) \n )\n\n (:action
 standup\n :parameters (?char - character)\n :precondition (or \
n (sitting ?char)\n (lying ?char)\n )\n :effect (and \n (not (
sitting ?char))\n (not (lying ?char))\n )\n )\n\n \n (:action
switchoff\n :parameters (?char - character ?obj - object)\n :
precondition (and\n (has_switch ?obj)\n (on ?obj)\n (next_to ?
char ?obj) \n ) \n :effect (and\n (off ?obj)\n (not (on ?obj))\n
 ) \n )\n \n (:action switchon\n :parameters (?char - character
?obj - object)\n :precondition (and\n (has_switch ?obj)\n (off ?
obj)\n (plugged_in ?obj)\n (next_to ?char ?obj)\n \n ) \n :
effect (and\n (on ?obj)\n (not (off ?obj))\n ) \n )\n\n (:action
 touch \n :parameters (?char - character ?obj - object)\n :
precondition (or\n (and \n (readable ?obj) \n (holds_lh ?char ?
obj)\n (not (exists (?obj2 - object) (and (obj_inside ?obj ?obj2
) (closed ?obj2))))\n )\n (and \n (readable ?obj) \n (holds_rh ?
char ?obj)\n (not (exists (?obj2 - object) (and (obj_inside ?obj
 ?obj2) (closed ?obj2))))\n )\n )\n :effect (and)\n )\n \n (:
action turnto\n :parameters (?char - character ?obj - object)\n
:precondition (and\n (not (exists (?obj2 - object) (and (
obj_inside ?obj ?obj2) (closed ?obj2))))\n ) \n :effect (and\n (
facing ?char ?obj)\n (forall (?obj2 - object) \n (when (not (and
 (obj_inside ?obj1 ?obj2) (open ?obj2))) \n (not (facing ?char ?
obj2))\n )\n )\n )\n )\n\n (:action watch \n :parameters (?char
- character ?obj - object)\n :precondition (and \n (lookable ?
obj) \n (facing ?char ?obj)\n )\n :effect (and\n (did_watch ?
```

```
char ?obj)\n )\n )\n \n (:action wipe\n :parameters (?char -
character ?obj1 - object ?obj2 - object)\n :precondition (or\n (
and \n (next_to ?char ?obj1) \n (holds_lh ?char ?obj2)\n )\n (
and \n (next_to ?char ?obj1) \n (holds_rh ?char ?obj2)\n )\n )\n
 :effect (and \n (clean ?obj1)\n (not (dirty ?obj1))\n )\n )\n\n
 (:action obj_puton\n :parameters (?char - character ?obj1 -
object ?obj2 - object)\n :precondition (or\n (and\n (next_to ?
char ?obj2)\n (holds_lh ?char ?obj1)\n )\n (and\n (next_to ?char
 ?obj2)\n (holds_rh ?char ?obj1)\n )\n )\n :effect (and\n (
obj_next_to ?obj1 ?obj2)\n (obj_ontop ?obj1 ?obj2)\n (not (
holds_lh ?char ?obj1))\n (not (holds_rh ?char ?obj1))\n )\n )\n
(:action puton\n :parameters (?char - character ?obj - object)\n
 :precondition (or\n (holds_lh ?char ?obj)\n (holds_rh ?char ?
obj)\n )\n :effect (and\n (on_char ?obj ?char)\n (not (holds_lh
?char ?obj))\n (not (holds_rh ?char ?obj))\n )\n )\n\n (:action
putoff_lh\n :parameters (?char - character ?obj - object)\n :
precondition (and\n (on_char ?obj ?char)\n (not (holds_lh ?char
?obj))\n )\n :effect (and\n (not (on_char ?obj ?char))\n (
holds_lh ?char ?obj)\n )\n )\n\n (:action putoff_rh\n :
parameters (?char - character ?obj - object)\n :precondition (
and\n (on_char ?obj ?char)\n (not (holds_rh ?char ?obj))\n )\n :
effect (and\n (not (on_char ?obj ?char))\n (holds_rh ?char ?obj)
\n )\n )\n\n (:action drop\n :parameters (?char - character ?obj
 - object ?room - object)\n :precondition (or\n (and \n (
holds_lh ?char ?obj)\n (obj_inside ?obj ?room)\n )\n (and \n (
holds_rh ?char ?obj)\n (obj_inside ?obj ?room)\n )\n ) \n :
effect (and\n (not (holds_lh ?char ?obj))\n (not (holds_rh ?char
 ?obj))\n ) \n )\n\n (:action read \n :parameters (?char -
character ?obj - object)\n :precondition (or\n (and \n (readable
 ?obj) \n (holds_lh ?char ?obj)\n )\n (and \n (readable ?obj) \n
 (holds_rh ?char ?obj)\n )\n )\n :effect (and)\n )\n \n (:action
 lie \n :parameters (?char - character ?obj - object)\n :
precondition (and \n (lieable ?obj) \n (next_to ?char ?obj)\n (
not (lying ?char))\n )\n :effect (and\n (lying ?char)\n (ontop ?
char ?obj)\n (not (sitting ?char))\n )\n )\n\n (:action pour \n
:parameters (?char - character ?obj1 - object ?obj2 - object)\n
:precondition (or\n (and \n (pourable ?obj1) \n (holds_lh ?char
?obj1)\n (recipient ?obj2)\n (next_to ?char ?obj2)\n )\n (and \n
 (pourable ?obj1) \n (holds_rh ?char ?obj1)\n (recipient ?obj2)\
n (next_to ?char ?obj2)\n )\n (and \n (drinkable ?obj1) \n (
holds_lh ?char ?obj1)\n (recipient ?obj2)\n (next_to ?char ?obj2
)\n )\n (and \n (drinkable ?obj1) \n (holds_rh ?char ?obj1)\n (
recipient ?obj2)\n (next_to ?char ?obj2)\n ) \n )\n :effect (
obj_inside ?obj1 ?obj2)\n )\n\n (:action type \n :parameters (?
char - character ?obj - object)\n :precondition (and \n (
has_switch ?obj) \n (next_to ?char ?obj)\n )\n :effect (and)\n )
\n\n (:action move \n :parameters (?char - character ?obj -
object)\n :precondition (and \n (movable ?obj) \n (next_to ?char
 ?obj)\n (not (exists (?obj2 - object) (and (obj_inside ?obj ?
obj2) (closed ?obj2))))\n )\n :effect (and)\n )\n\n (:action
wash \n :parameters (?char - character ?obj - object)\n :
precondition (and \n (next_to ?char ?obj)\n )\n :effect (and\n (
clean ?obj)\n (not (dirty ?obj))\n )\n )\n\n (:action squeeze \n
 :parameters (?char - character ?obj - object)\n :precondition (
and \n (next_to ?char ?obj)\n (clothes ?obj)\n )\n :effect (and)
\n )\n\n (:action plugin \n :parameters (?char - character ?obj
- object)\n :precondition (or\n (and \n (next_to ?char ?obj)\n (
```

```
    has_plug ?obj)\n (plugged_out ?obj)\n )\n (and \n (next_to ?char
     ?obj)\n (has_switch ?obj)\n (plugged_out ?obj)\n )\n )\n :
    effect (and\n (plugged_in ?obj)\n (not (plugged_out ?obj))\n )\n
     )\n\n (:action plugout \n :parameters (?char - character ?obj -
     object)\n :precondition (and \n (next_to ?char ?obj)\n (
    has_plug ?obj)\n (plugged_in ?obj)\n (not (on ?obj))\n )\n :
    effect (and\n (plugged_out ?obj)\n (not (plugged_in ?obj))\n )\n
     )\n\n (:action cut \n :parameters (?char - character ?obj -
    object)\n :precondition (and \n (next_to ?char ?obj)\n (eatable
    ?obj)\n (cuttable ?obj)\n )\n :effect (and)\n )\n\n (:action eat
     \n :parameters (?char - character ?obj - object)\n :
    precondition (and \n (next_to ?char ?obj)\n (eatable ?obj)\n )\n
     :effect (and)\n )\n\n (:action sleep \n :parameters (?char -
    character ?obj - object)\n :precondition (or \n (lying ?char)\n
    (sitting ?char)\n )\n :effect (and)\n )\n\n (:action wake_up \n
    :parameters (?char - character ?obj - object)\n :precondition (
    or \n (lying ?char)\n (sitting ?char)\n )\n :effect (and)\n )\n)
    ",

"corrected_problem": "(define (problem generated)\n (: domain
    virtualhome)\n (:objects\n character_id65 - character\n
    couch_id352 table_id355 remote_control_id1000 dining_room_id201
    television_id410 home_office_id319 - object\n )\n (:init\n (
    clean television_id410)\n (grabbable remote_control_id1000)\n (
    has_plug television_id410)\n (has_switch remote_control_id1000)\
    n (has_switch television_id410)\n (inside_room character_id65
    dining_room_id201)\n (lieable couch_id352)\n (lookable
    television_id410)\n (movable couch_id352)\n (movable
    remote_control_id1000)\n (movable table_id355)\n (obj_next_to
    couch_id352 remote_control_id1000)\n (obj_next_to
    remote_control_id1000 couch_id352)\n (off television_id410)\n (
    plugged_in television_id410)\n (sittable couch_id352)\n (
    surfaces couch_id352)\n (surfaces table_id355)\n )\n (:goal\n (
    and\n (facing character_id65 television_id410)\n (on
    television_id410)\n (plugged_in television_id410)\n (did_watch
    character_id65 television_id410)\n )\n )\n)\n",

"action_plan": [\"(turnto character_id65 television_id410)\", \"(
    walk character_id65 television_id410)\", \"(switchon
    character_id65 television_id410)\", \"(watch character_id65
    television_id410)\"]
}}

## Explanation (not part of the output):
To incorporate the action goals with either actions LOOKAT or WATCH,
     we add two new predicates (did_watch ?char - character ?obj -
    object) and (did_lookat ?char - character ?obj - object) to the
    domain file.Then we add to the effect of (:action watch) the (
    did_watch ?char ?obj) predicate and add to the effect of (:
    action lookat) the (did_lookat ?char ?obj) predicate. This
    allows us to add to the (:goal) of the problem (did_watch
    character_id65 television_id410), which is inferred from the
    required action goal and the natural language command.



Now, it is your turn to correct and solve the following PDDL
```

```
    problem. Comprehend the action goals and correct the main as
    necessary. Correct the resulted domain and the problem. Finally,
     solve the problem with an action plan. Remember the action plan
     needs at least one action. No explanation is needed, and the
    output marker is provided. Strictly adhere to the OUTPUT FORMAT
    and generate NOTHING ELSE.

[Natural Language Command]
{nlp_command}

---
[Action Goals]
{action_goals}

---
[PDDL Problem File
{problem_file}

---
OUTPUT:
```

D.4 Transition Modeling

D.4.1 BEHAVIOR

Template 20: BEHAVIOR – Transition Modeling Original Template

```
The following is predicates defined in this domain file. Pay
    attention to the types for each predicate.
(define (domain igibson)

   (:requirements :strips :adl :typing :negative-preconditions)

   (:types
      vacuum_n_04 facsimile_n_02 dishtowel_n_01 apparel_n_01
          seat_n_03 bottle_n_01 mouse_n_04 window_n_01 scanner_n_02
      sauce_n_01 spoon_n_01 date_n_08 egg_n_02 cabinet_n_01
          yogurt_n_01 parsley_n_02 notebook_n_01 dryer_n_01
          saucepan_n_01
      soap_n_01 package_n_02 headset_n_01 fish_n_02 vehicle_n_01
          chestnut_n_03 grape_n_01 wrapping_n_01 makeup_n_01
          mug_n_04
      pasta_n_02 beef_n_02 scrub_brush_n_01 cracker_n_01 flour_n_01
           sunglass_n_01 cookie_n_01 bed_n_01 lamp_n_02 food_n_02
      painting_n_01 carving_knife_n_01 pop_n_02 tea_bag_n_01
          sheet_n_03 tomato_n_01 agent_n_01 hat_n_01 dish_n_01
          cheese_n_01
      perfume_n_02 toilet_n_02 broccoli_n_02 book_n_02 towel_n_01
          table_n_02 pencil_n_01 rag_n_01 peach_n_03 water_n_06
          cup_n_01
      radish_n_01 marker_n_03 tile_n_01 box_n_01 screwdriver_n_01
          raspberry_n_02 banana_n_02 grill_n_02 caldron_n_01
          vegetable_oil_n_01
      necklace_n_01 brush_n_02 washer_n_03 hamburger_n_01
          catsup_n_01 sandwich_n_01 plaything_n_01 candy_n_01
```

```
        cereal_n_03 door_n_01
      food_n_01 newspaper_n_03 hanger_n_02 carrot_n_03 salad_n_01
        toothpaste_n_01 blender_n_01 sofa_n_01 plywood_n_01
        olive_n_04 briefcase_n_01
      christmas_tree_n_05 bowl_n_01 casserole_n_02 apple_n_01
        basket_n_01 pot_plant_n_01 backpack_n_01 sushi_n_01
        saw_n_02 toothbrush_n_01
      lemon_n_01 pad_n_01 receptacle_n_01 sink_n_01 countertop_n_01
         melon_n_01 bracelet_n_02 modem_n_01 pan_n_01 oatmeal_n_01
         calculator_n_02
      duffel_bag_n_01 sandal_n_01 floor_n_01 snack_food_n_01
        stocking_n_01 dishwasher_n_01 pencil_box_n_01 chicken_n_01
         jar_n_01 alarm_n_02
      stove_n_01 plate_n_04 highlighter_n_02 umbrella_n_01
        piece_of_cloth_n_01 bin_n_01 ribbon_n_01 chip_n_04
        shelf_n_01 bucket_n_01 shampoo_n_01
      folder_n_02 shoe_n_01 detergent_n_02 milk_n_01 beer_n_01
        shirt_n_01 dustpan_n_02 cube_n_05 broom_n_01 candle_n_01
        pen_n_01 microwave_n_02
      knife_n_01 wreath_n_01 car_n_01 soup_n_01 sweater_n_01
        tray_n_01 juice_n_01 underwear_n_01 orange_n_01
        envelope_n_01 fork_n_01 lettuce_n_03
      bathtub_n_01 earphone_n_01 pool_n_01 printer_n_03 sack_n_01
        highchair_n_01 cleansing_agent_n_01 kettle_n_01
        vidalia_onion_n_01 mousetrap_n_01
      bread_n_01 meat_n_01 mushroom_n_05 cake_n_03 vessel_n_03
        bow_n_08 gym_shoe_n_01 hammer_n_02 teapot_n_01 chair_n_01
        jewelry_n_01 pumpkin_n_02 sugar_n_01
      shower_n_01 ashcan_n_01 hand_towel_n_01 pork_n_01
        strawberry_n_01 electric_refrigerator_n_01 oven_n_01
        ball_n_01 document_n_01 sock_n_01 beverage_n_01
      hardback_n_01 scraper_n_01 carton_n_02
      agent
   )

   (:predicates
      (inside ?obj1 - object ?obj2 - object)
      (nextto ?obj1 - object ?obj2 - object)
      (ontop ?obj1 - object ?obj2 - object)
      (under ?obj1 - object ?obj2 - object)
      (cooked ?obj1 - object)
      (dusty ?obj1 - object)
      (frozen ?obj1 - object)
      (open ?obj1 - object)
      (stained ?obj1 - object)
      (sliced ?obj1 - object)
      (soaked ?obj1 - object)
      (toggled_on ?obj1 - object)
      (onfloor ?obj1 - object ?floor1 - object)
      (holding ?obj1 - object)
      (handsfull ?agent1 - agent)
      (in_reach_of_agent ?obj1 - object)
      (same_obj ?obj1 - object ?obj2 - object)
   )
   ;; Actions to be predicted
)
Objective: Given the problem file of pddl, which defines objects in
```

```
     the task (:objects), initial conditions (:init) and goal
    conditions (:goal), write the body of PDDL actions (:
    precondition and :effect) given specific action names and
    parameters.
Each PDDL action definition consists of four main components:
    action name, parameters, precondition, and effect. Here is the
    general format to follow:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)
The :parameters is the list of variables on which the action
    operates. It lists variable names and variable types.
The :precondition is a first-order logic sentence specifying
    preconditions for an action. The precondition consists of
    predicates and 3 possible logical operators: or, and, and not.
    The precondition should be structured in Disjunctive Normal Form
     (DNF), meaning an OR of ANDs. The not operator should only be
    used within these conjunctions. For example, (or (and (
    predicate1 ?x) (predicate2 ?y)) (and (predicate3 ?x)))
The :effect lists the changes which the action imposes on the
    current state. The precondition consists of predicates and 3
    possible logical operators: and, not and when. 1. The effects
    should generally be several effects connected by AND operators.
    2. For each effect, if it is a conditional effect, use WHEN to
    check the conditions. The semantics of (when [condition] [effect
    ]) are as follows: If [condition] is true before the action,
    then [effect] occurs afterwards. 3. If it is not a conditional
    effect, use predicates directly. 4. The NOT operator is used to
    negate a predicate, signifying that the condition will not hold
    after the action is executed. And example of effect is (and (
    when (predicate1 ?x) (not (predicate2 ?y))) (predicate3 ?x))
In any case, the occurrence of a predicate should agree with its
    declaration in terms of number and types of arguments defined in
     DOMAIN FILE at the beginning.

hint:
1. In many cases WHEN is not necessary. Don't enforce the use of
    WHEN
2. You MUST only use predicates and object types exactly as they
    appear in the domain file at the beginning. Now given the input,
     please fill in the action body for each provided actions in
    PDDL format.
For actions to be finished, write their preconditions and effects,
    and return in standard PDDL format:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)
Concatenate all actions PDDL string into a single string. Output in
     json format where key is "output" and value is your output
    string: {{"output": YOUR OUTPUT STRING}}

Here is an example of the input problem file and unfinished action:
Input:
```

```
Problem file:
(define (problem cleaning_floor_0)
   (:domain igibson)

   (:objects
      floor_n_01_1 - floor_n_01
      rag_n_01_1 - rag_n_01
      sink_n_01_1 - sink_n_01
      agent_n_01_1 - agent_n_01
   )

   (:init
      (dusty floor_n_01_1)
      (stained floor_n_01_2)
      (ontop rag_n_01_1 table_n_02_1)
      (inroom sink_n_01_1 storage_room)
      (onfloor agent_n_01_1 floor_n_01_2)
   )

   (:goal
      (and
      (not (dusty floor_n_01_1))
      (not (stained floor_n_01_2))
      )
   )
)
Action to be finished:
(:action navigate_to
      :parameters (?objto - object ?agent - agent)
      :precondition ()
      :effect ()
   )
(:action clean-stained-floor-rag
  :parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent -
      agent_n_01)
  :precondition ()
  :effect ()
)
(:action clean-dusty-floor-rag
  :parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent -
      agent_n_01)
  :precondition ()
  :effect ()
)
(:action soak-rag
  :parameters (?rag - rag_n_01 ?sink - sink_n_01 ?agent - agent_n_01
      )
  :precondition ()
  :effect ()
)
Output:
{{"output":
"(:action navigate_to_and_grasp
      :parameters (?objto - object ?agent - agent)
      :precondition (not (holding ?objto))
      :effect (and (holding ?objto)
                (in_reach_of_agent ?objto)
```

```
                (forall
                   (?objfrom - object)
                   (when
                      (and
                       (or
                         (in_reach_of_agent ?objfrom)
                         (holding ?objfrom)
                        )
                        (not (same_obj ?objfrom ?objto))
                      )
                      (and
                        (not (holding ?objfrom))
                        (not (in_reach_of_agent ?objfrom))
                      )
                   )
                )
             )
  )
 (:action clean-stained-floor-rag
  :parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent -
      agent_n_01)
  :precondition (and
             (stained ?floor)
             (soaked ?rag)
             (holding ?rag))
  :effect (and
        (not (stained ?floor))
        (in_reach_of_agent ?floor)
       )

 (:action clean-dusty-floor-rag
  :parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent -
      agent_n_01)
  :precondition (and
                 (dusty ?obj)
                 (holding ?rag))
  :effect (and
        (not (dusty ?floor))
        (in_reach_of_agent ?floor)
       )
)
 (:action soak-rag
  :parameters (?rag - rag_n_01 ?sink - sink_n_01 ?agent - agent_n_01
      )
  :precondition (and (holding ?rag)
                 (in_reach_of_agent ?sink)
                 (toggled_on ?sink))
  :effect (soaked ?rag)

 (:action reach
  :parameters (?obj - object ?agent - agent_n_01)
  :precondition (and (not (in_reach_of_agent ?obj)))
  :effect (and (in_reach_of_agent ?obj)
                (forall
                   (?objfrom - object)
                   (when
                      (and
```

```
                        (in_reach_of_agent ?objfrom)
                        (not (same_obj ?objfrom ?objto))
                    )
                    (not (in_reach_of_agent ?objfrom))
                )
            )
        )
)"}}

Input:
Problem file:
{problem_file}
Action to be finished:
{action_handler}
Output:
```

Template 21: BEHAVIOR – Transition Modeling Updated Template (v1)

```
Your task is to act as an expert PDDL (Planning Domain Definition
    Language) solver and designer. Your goal is to define the
    preconditions and effects for a set of incomplete action
    skeletons. You must use the domain's predicates to build a
    logical chain that connects the problem's initial state to its
    goal state. Once the actions are defined, you must output a
    valid plan to achieve the goal.
The general strategy you might use is
1. Analyze the Problem (Parse Input)
2. Work Backward from the Goal (Define Effects). The easiest way to
     start is by looking at the :goal and mapping it to an action.
3. Define Preconditions (The "Why"). Now that you have an effect,
    ask: "What must be true for this action to happen?"
for examples: Obvious Preconditions: the state must be the opposite
     of the effect, the agent must be present, the agent must have
    the tool; Inferential Preconditions (The "Key Logic"): Look at
    the other actions. Why are there two cleaning actions (clean-
    dusty and clean-stained)? Find a predicate that captures this
    difference. The (soaked ?rag) predicate is the key.
4. Recurse on New Subgoals. You just created new subgoals (
    preconditions). Find the action that achieves this subgoal and
    repeat the same process.
5. Handle State Changes and Deletes (Rigor). Actions don't just add
     facts; they also delete them.
6. Assemble the Final Plan. Finally, trace the complete logical
    chain you have built, starting from the :init state. If the plan
     is not reachable, rework the action definitions again.

The following is predicates defined in this domain file. Pay
    attention to the types for each predicate.
(define (domain igibson)

   (:requirements :strips :adl :typing :negative-preconditions)

   (:types
      vacuum_n_04 facsimile_n_02 dishtowel_n_01 apparel_n_01
          seat_n_03 bottle_n_01 mouse_n_04 window_n_01 scanner_n_02
```

```
    sauce_n_01 spoon_n_01 date_n_08 egg_n_02 cabinet_n_01
        yogurt_n_01 parsley_n_02 notebook_n_01 dryer_n_01
        saucepan_n_01
    soap_n_01 package_n_02 headset_n_01 fish_n_02 vehicle_n_01
        chestnut_n_03 grape_n_01 wrapping_n_01 makeup_n_01
        mug_n_04
    pasta_n_02 beef_n_02 scrub_brush_n_01 cracker_n_01 flour_n_01
         sunglass_n_01 cookie_n_01 bed_n_01 lamp_n_02 food_n_02
    painting_n_01 carving_knife_n_01 pop_n_02 tea_bag_n_01
        sheet_n_03 tomato_n_01 agent_n_01 hat_n_01 dish_n_01
        cheese_n_01
    perfume_n_02 toilet_n_02 broccoli_n_02 book_n_02 towel_n_01
        table_n_02 pencil_n_01 rag_n_01 peach_n_03 water_n_06
        cup_n_01
    radish_n_01 marker_n_03 tile_n_01 box_n_01 screwdriver_n_01
        raspberry_n_02 banana_n_02 grill_n_02 caldron_n_01
        vegetable_oil_n_01
    necklace_n_01 brush_n_02 washer_n_03 hamburger_n_01
        catsup_n_01 sandwich_n_01 plaything_n_01 candy_n_01
        cereal_n_03 door_n_01
    food_n_01 newspaper_n_03 hanger_n_02 carrot_n_03 salad_n_01
        toothpaste_n_01 blender_n_01 sofa_n_01 plywood_n_01
        olive_n_04 briefcase_n_01
    christmas_tree_n_05 bowl_n_01 casserole_n_02 apple_n_01
        basket_n_01 pot_plant_n_01 backpack_n_01 sushi_n_01
        saw_n_02 toothbrush_n_01
    lemon_n_01 pad_n_01 receptacle_n_01 sink_n_01 countertop_n_01
         melon_n_01 bracelet_n_02 modem_n_01 pan_n_01 oatmeal_n_01
         calculator_n_02
    duffel_bag_n_01 sandal_n_01 floor_n_01 snack_food_n_01
        stocking_n_01 dishwasher_n_01 pencil_box_n_01 chicken_n_01
         jar_n_01 alarm_n_02
    stove_n_01 plate_n_04 highlighter_n_02 umbrella_n_01
        piece_of_cloth_n_01 bin_n_01 ribbon_n_01 chip_n_04
        shelf_n_01 bucket_n_01 shampoo_n_01
    folder_n_02 shoe_n_01 detergent_n_02 milk_n_01 beer_n_01
        shirt_n_01 dustpan_n_02 cube_n_05 broom_n_01 candle_n_01
        pen_n_01 microwave_n_02
    knife_n_01 wreath_n_01 car_n_01 soup_n_01 sweater_n_01
        tray_n_01 juice_n_01 underwear_n_01 orange_n_01
        envelope_n_01 fork_n_01 lettuce_n_03
    bathtub_n_01 earphone_n_01 pool_n_01 printer_n_03 sack_n_01
        highchair_n_01 cleansing_agent_n_01 kettle_n_01
        vidalia_onion_n_01 mousetrap_n_01
    bread_n_01 meat_n_01 mushroom_n_05 cake_n_03 vessel_n_03
        bow_n_08 gym_shoe_n_01 hammer_n_02 teapot_n_01 chair_n_01
        jewelry_n_01 pumpkin_n_02 sugar_n_01
    shower_n_01 ashcan_n_01 hand_towel_n_01 pork_n_01
        strawberry_n_01 electric_refrigerator_n_01 oven_n_01
        ball_n_01 document_n_01 sock_n_01 beverage_n_01
    hardback_n_01 scraper_n_01 carton_n_02
    agent
)

(:predicates
    (inside ?obj1 - object ?obj2 - object)
    (nextto ?obj1 - object ?obj2 - object)
```

```
      (ontop ?obj1 - object ?obj2 - object)
      (under ?obj1 - object ?obj2 - object)
      (cooked ?obj1 - object)
      (dusty ?obj1 - object)
      (frozen ?obj1 - object)
      (open ?obj1 - object)
      (stained ?obj1 - object)
      (sliced ?obj1 - object)
      (soaked ?obj1 - object)
      (toggled_on ?obj1 - object)
      (onfloor ?obj1 - object ?floor1 - object)
      (holding ?obj1 - object)
      (handsfull ?agent1 - agent)
      (in_reach_of_agent ?obj1 - object)
      (same_obj ?obj1 - object ?obj2 - object)
   )
   ;; Actions to be predicted
)
Objective: Given the problem file of pddl, which defines objects in
     the task (:objects), initial conditions (:init) and goal
    conditions (:goal), write the body of PDDL actions (:
    precondition and :effect) given specific action names and
    parameters.
Each PDDL action definition consists of four main components:
    action name, parameters, precondition, and effect. Here is the
    general format to follow:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)
The :parameters is the list of variables on which the action
    operates. It lists variable names and variable types.
The :precondition is a first-order logic sentence specifying
    preconditions for an action. The precondition consists of
    predicates and 3 possible logical operators: or, and, and not.
    The precondition should be structured in Disjunctive Normal Form
     (DNF), meaning an OR of ANDs. The not operator should only be
    used within these conjunctions. For example, (or (and (
    predicate1 ?x) (predicate2 ?y)) (and (predicate3 ?x)))
The :effect lists the changes which the action imposes on the
    current state. The precondition consists of predicates and 3
    possible logical operators: and, not and when. 1. The effects
    should generally be several effects connected by AND operators.
    2. For each effect, if it is a conditional effect, use WHEN to
    check the conditions. The semantics of (when [condition] [effect
    ]) are as follows: If [condition] is true before the action,
    then [effect] occurs afterwards. 3. If it is not a conditional
    effect, use predicates directly. 4. The NOT operator is used to
    negate a predicate, signifying that the condition will not hold
    after the action is executed. And example of effect is (and (
    when (predicate1 ?x) (not (predicate2 ?y))) (predicate3 ?x))
In any case, the occurrence of a predicate should agree with its
    declaration in terms of number and types of arguments defined in
     DOMAIN FILE at the beginning.

hint:
```

```
1. In many cases WHEN is not necessary. Don't enforce the use of
    WHEN
2. You MUST only use predicates and object types exactly as they
    appear in the domain file at the beginning. Now given the input,
     please fill in the action body for each provided actions in
    PDDL format.
For actions to be finished, write their preconditions and effects,
    and return in standard PDDL format:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)

OUTPUT FORMAT:
Concatenate all actions PDDL string into a single string. Output in
     json format where key is "output" and value is your output
    string.
Put all PDDL action strings of the final plan into a JSON list of
    strings.
{{
"output": "YOUR OUTPUT STRING",
"action_plan": ["YOUR", "ACTION", "PLAN"]
}}

Here is an example of the input problem file and unfinished action:
Input:
Problem file:
(define (problem cleaning_floor_0)
   (:domain igibson)

   (:objects
      floor_n_01_1 - floor_n_01
      floor_n_01_2 - floor_n_01
      rag_n_01_1 - rag_n_01
      sink_n_01_1 - sink_n_01
      agent_n_01_1 - agent
   )

   (:init
      (dusty floor_n_01_1)
      (stained floor_n_01_2)
      (ontop rag_n_01_1 table_n_02_1)
      (inroom sink_n_01_1 storage_room)
      (onfloor agent_n_01_1 floor_n_01_2)
   )

   (:goal
      (and
      (not (dusty floor_n_01_1))
      (not (stained floor_n_01_2))
      )
   )
)
Action to be finished:
(:action navigate_to_and_grasp
      :parameters (?objto - object ?agent - agent)
```

```
      :precondition ()
      :effect ()
   )
(:action clean-stained-floor-rag
  :parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent - agent)
  :precondition ()
  :effect ()
)
(:action clean-dusty-floor-rag
  :parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent - agent)
  :precondition ()
  :effect ()
)
(:action soak-rag
  :parameters (?rag - rag_n_01 ?sink - sink_n_01 ?agent - agent)
  :precondition ()
  :effect ()
)
(:action reach
  :parameters (?obj - object ?agent - agent)
  :precondition ()
  :effect ()
)
Output:
{{"output":
"(:action navigate_to_and_grasp
      :parameters (?objto - object ?agent - agent)
      :precondition (not (holding ?objto))
      :effect (and (holding ?objto)
                (in_reach_of_agent ?objto)
                (forall
                   (?objfrom - object)
                   (when
                      (and
                        (or
                          (in_reach_of_agent ?objfrom)
                          (holding ?objfrom)
                        )
                        (not (same_obj ?objfrom ?objto))
                      )
                      (and
                        (not (holding ?objfrom))
                        (not (in_reach_of_agent ?objfrom))
                      )
                   )
                )
            )
 )
(:action clean-stained-floor-rag
  :parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent - agent)
  :precondition (and
             (stained ?floor)
             (soaked ?rag)
             (holding ?rag))
  :effect (and
        (not (stained ?floor))
        (in_reach_of_agent ?floor)
```

```
      )

(:action clean-dusty-floor-rag
  :parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent - agent)
  :precondition (and
                 (dusty ?obj)
                 (holding ?rag))
  :effect (and
        (not (dusty ?floor))
        (in_reach_of_agent ?floor)
      )
)
(:action soak-rag
  :parameters (?rag - rag_n_01 ?sink - sink_n_01 ?agent - agent)
  :precondition (and (holding ?rag)
                 (in_reach_of_agent ?sink)
                 (toggled_on ?sink))
  :effect (soaked ?rag)

(:action reach
  :parameters (?obj - object ?agent - agent)
  :precondition (and (not (in_reach_of_agent ?obj)))
  :effect (and (in_reach_of_agent ?obj)
               (forall
                  (?objfrom - object)
                  (when
                     (and
                        (in_reach_of_agent ?objfrom)
                        (not (same_obj ?objfrom ?objto))
                     )
                     (not (in_reach_of_agent ?objfrom))
                  )
               )
        )
)",
"action_plan": [\"(navigate_to_and_grasp rag_n_01_1 agent_n_01_1)\",
     \"(reach floor_n_01_1 agent_n_01_1)\", \"(clean-dusty-floor-rag
     rag_n_01_1 floor_n_01_1 agent_n_01_1)\", \"(reach sink_n_01_1
    agent_n_01_1)\", \"(soak-rag rag_n_01_1 sink_n_01_1 agent_n_01_1
    )\", \"(reach floor_n_01_2 agent_n_01_1)\", \"(clean-stained-
    floor-rag rag_n_01_1 floor_n_01_2 agent_n_01_1)\"]
}}

Explanation (not part of the output):
We began by analyzing the goal: `(not (dusty floor_n_01_1))` and `(
    not (stained floor_n_01_2))`. The presence of two distinct
    actions, `clean-dusty-floor-rag` and `clean-stained-floor-rag`,
    was the key insight. We inferred this distinction meant cleaning
     a stain requires a wet rag, while cleaning dust requires a dry
    one. We used a simple real-world logic to connect them: we clean
     dust with a dry rag, but we clean a stain with a wet (soaked)
    rag. We used the domain's `(soaked)` predicate to model this.
This led us to define the core cleaning actions:
1. `clean-stained-floor-rag`: Its main precondition is `(soaked ?
    rag)` and its effect is `(not (stained ?floor))`.
2. `clean-dusty-floor-rag`: Its main precondition is `(not (soaked
    ?rag))` and its effect is `(not (dusty ?floor))`.
```

This created a logical chain. To satisfy the `(soaked ?rag)` precondition, we defined the `soak-rag` action. Its primary effect is `(soaked ?rag)`, and its preconditions are that the agent is `(holding ?rag)` and `(in_reach_of_agent ?sink)`.
To prevent a deadlock, we modify the initial state so the rag must start in a known state. We add (not (soaked rag_n_01_1)) to the (:init) block.
To satisfy `(holding ?rag)`, we defined `navigate_to_and_grasp`. Its effects are to make the agent `(holding ?objto)` and `(handsfull ?agent)`, while also deleting the rag's initial `(ontop ...)` predicate.
Finally, we needed to manage the agent's location. We defined `reach` as the navigation action. Its effect is `(in_reach_of_agent ?obj)`. Critically, we added a `(forall ...)` effect to both `reach` and `navigate_to_and_grasp` to remove the agent's *previous* `(in_reach_of_agent ?other)` predicate. This ensures the agent is only in one location at a time. We also added `(not (in_reach_of_agent ?obj))` as a precondition to `reach` to prevent redundant moves.

This set of definitions built the 7-step plan:
1. The agent must first `navigate_to_and_grasp` the rag (which is initially dry).
2. With a dry rag, the agent
(a) `reach` the dusty floor and satisfy the precondition for `clean-dusty-floor-rag`.
(b) Execute `clean-dusty-floor-rag` induce the effect of the first goal.
3. The agent must then
(a) `reach` the sink and
(b) `soak-rag` to make it wet.
4. With a wet rag, the agent can
(a) `reach` the stained floor and and satisfy the precondition for `clean-dusty-floor-rag`.
(b) Execute `clean-stained-floor-rag` induce the effect of the second goal. This final effect allows the agent to thus completing the overall goal.

Now it's your turn. Define the preconditions and effects for a set of incomplete action skeletons, then output the valid plan to achieve the goal. Strictly format your output using the JSON from the OUTPUT FORMAT, no prefix, suffix, or explanation is needed.

Input:
Problem file:
{problem_file}
Action to be finished:
{action_handler}
Output:

Template 22: BEHAVIOR – Transition Modeling Updated Template (v2)

```
Your task is to act as an expert PDDL (Planning Domain Definition
    Language) solver and designer. Your goal is to define the
    preconditions and effects for a set of incomplete action
    skeletons. You must use the domain's predicates to build a
    logical chain that connects the problem's initial state to its
    goal state. Once the actions are defined, you must output a
    valid plan to achieve the goal.
The general strategy you might use is
1. Analyze the Problem (Parse Input)
2. Work Backward from the Goal (Define Effects). The easiest way to
     start is by looking at the :goal and mapping it to an action.
3. Define Preconditions (The "Why"). Now that you have an effect,
    ask: "What must be true for this action to happen?"
for examples: Obvious Preconditions: the state must be the opposite
     of the effect, the agent must be present, the agent must have
    the tool; Inferential Preconditions (The "Key Logic"): Look at
    the other actions. Why are there two cleaning actions (clean-
    dusty and clean-stained)? Find a predicate that captures this
    difference. The (soaked ?rag) predicate is the key.
4. Recurse on New Subgoals. You just created new subgoals (
    preconditions). Find the action that achieves this subgoal and
    repeat the same process.
5. Handle State Changes and Deletes (Rigor). Actions don't just add
     facts; they also delete them.
6. Assemble the Final Plan. Finally, trace the complete logical
    chain you have built, starting from the :init state. If the plan
     is not reachable, rework the action definitions again.

The following is predicates defined in this domain file. Pay
    attention to the types for each predicate.
(define (domain igibson)

   (:requirements :strips :adl :typing :negative-preconditions)

   (:types
      vacuum_n_04 facsimile_n_02 dishtowel_n_01 apparel_n_01
          seat_n_03 bottle_n_01 mouse_n_04 window_n_01 scanner_n_02
      sauce_n_01 spoon_n_01 date_n_08 egg_n_02 cabinet_n_01
          yogurt_n_01 parsley_n_02 notebook_n_01 dryer_n_01
          saucepan_n_01
      soap_n_01 package_n_02 headset_n_01 fish_n_02 vehicle_n_01
          chestnut_n_03 grape_n_01 wrapping_n_01 makeup_n_01
          mug_n_04
      pasta_n_02 beef_n_02 scrub_brush_n_01 cracker_n_01 flour_n_01
           sunglass_n_01 cookie_n_01 bed_n_01 lamp_n_02 food_n_02
      painting_n_01 carving_knife_n_01 pop_n_02 tea_bag_n_01
          sheet_n_03 tomato_n_01 agent_n_01 hat_n_01 dish_n_01
          cheese_n_01
      perfume_n_02 toilet_n_02 broccoli_n_02 book_n_02 towel_n_01
          table_n_02 pencil_n_01 rag_n_01 peach_n_03 water_n_06
          cup_n_01
      radish_n_01 marker_n_03 tile_n_01 box_n_01 screwdriver_n_01
          raspberry_n_02 banana_n_02 grill_n_02 caldron_n_01
          vegetable_oil_n_01
      necklace_n_01 brush_n_02 washer_n_03 hamburger_n_01
```

```
        catsup_n_01 sandwich_n_01 plaything_n_01 candy_n_01
        cereal_n_03 door_n_01
    food_n_01 newspaper_n_03 hanger_n_02 carrot_n_03 salad_n_01
        toothpaste_n_01 blender_n_01 sofa_n_01 plywood_n_01
        olive_n_04 briefcase_n_01
    christmas_tree_n_05 bowl_n_01 casserole_n_02 apple_n_01
        basket_n_01 pot_plant_n_01 backpack_n_01 sushi_n_01
        saw_n_02 toothbrush_n_01
    lemon_n_01 pad_n_01 receptacle_n_01 sink_n_01 countertop_n_01
         melon_n_01 bracelet_n_02 modem_n_01 pan_n_01 oatmeal_n_01
         calculator_n_02
    duffel_bag_n_01 sandal_n_01 floor_n_01 snack_food_n_01
        stocking_n_01 dishwasher_n_01 pencil_box_n_01 chicken_n_01
         jar_n_01 alarm_n_02
    stove_n_01 plate_n_04 highlighter_n_02 umbrella_n_01
        piece_of_cloth_n_01 bin_n_01 ribbon_n_01 chip_n_04
        shelf_n_01 bucket_n_01 shampoo_n_01
    folder_n_02 shoe_n_01 detergent_n_02 milk_n_01 beer_n_01
        shirt_n_01 dustpan_n_02 cube_n_05 broom_n_01 candle_n_01
        pen_n_01 microwave_n_02
    knife_n_01 wreath_n_01 car_n_01 soup_n_01 sweater_n_01
        tray_n_01 juice_n_01 underwear_n_01 orange_n_01
        envelope_n_01 fork_n_01 lettuce_n_03
    bathtub_n_01 earphone_n_01 pool_n_01 printer_n_03 sack_n_01
        highchair_n_01 cleansing_agent_n_01 kettle_n_01
        vidalia_onion_n_01 mousetrap_n_01
    bread_n_01 meat_n_01 mushroom_n_05 cake_n_03 vessel_n_03
        bow_n_08 gym_shoe_n_01 hammer_n_02 teapot_n_01 chair_n_01
        jewelry_n_01 pumpkin_n_02 sugar_n_01
    shower_n_01 ashcan_n_01 hand_towel_n_01 pork_n_01
        strawberry_n_01 electric_refrigerator_n_01 oven_n_01
        ball_n_01 document_n_01 sock_n_01 beverage_n_01
    hardback_n_01 scraper_n_01 carton_n_02
    agent
)

(:predicates
    (inside ?obj1 - object ?obj2 - object)
    (nextto ?obj1 - object ?obj2 - object)
    (ontop ?obj1 - object ?obj2 - object)
    (under ?obj1 - object ?obj2 - object)
    (cooked ?obj1 - object)
    (dusty ?obj1 - object)
    (frozen ?obj1 - object)
    (open ?obj1 - object)
    (stained ?obj1 - object)
    (sliced ?obj1 - object)
    (soaked ?obj1 - object)
    (toggled_on ?obj1 - object)
    (onfloor ?obj1 - object ?floor1 - object)
    (holding ?obj1 - object)
    (handsfull ?agent1 - agent)
    (in_reach_of_agent ?obj1 - object)
    (same_obj ?obj1 - object ?obj2 - object)
)
;; Actions to be predicted
)
```

```
Objective: Given the problem file of pddl, which defines objects in
     the task (:objects), initial conditions (:init) and goal
    conditions (:goal), write the body of PDDL actions (:
    precondition and :effect) given specific action names and
    parameters.
Each PDDL action definition consists of four main components:
    action name, parameters, precondition, and effect. Here is the
    general format to follow:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)
The :parameters is the list of variables on which the action
    operates. It lists variable names and variable types.
The :precondition is a first-order logic sentence specifying
    preconditions for an action. The precondition consists of
    predicates and 3 possible logical operators: or, and, and not.
    The precondition should be structured in Disjunctive Normal Form
     (DNF), meaning an OR of ANDs. The not operator should only be
    used within these conjunctions. For example, (or (and (
    predicate1 ?x) (predicate2 ?y)) (and (predicate3 ?x)))
The :effect lists the changes which the action imposes on the
    current state. The precondition consists of predicates and 3
    possible logical operators: and, not and when. 1. The effects
    should generally be several effects connected by AND operators.
    2. For each effect, if it is a conditional effect, use WHEN to
    check the conditions. The semantics of (when [condition] [effect
    ]) are as follows: If [condition] is true before the action,
    then [effect] occurs afterwards. 3. If it is not a conditional
    effect, use predicates directly. 4. The NOT operator is used to
    negate a predicate, signifying that the condition will not hold
    after the action is executed. And example of effect is (and (
    when (predicate1 ?x) (not (predicate2 ?y))) (predicate3 ?x))
In any case, the occurrence of a predicate should agree with its
    declaration in terms of number and types of arguments defined in
     DOMAIN FILE at the beginning.

hint:
1. In many cases WHEN is not necessary. Don't enforce the use of
    WHEN
2. You MUST only use predicates and object types exactly as they
    appear in the domain file at the beginning. Now given the input,
     please fill in the action body for each provided actions in
    PDDL format.
For actions to be finished, write their preconditions and effects,
    and return in standard PDDL format:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)

Here are some commonly used actions:
(:action navigate_to
   :parameters (?objto - object ?agent - agent)
   :precondition (not (in_reach_of_agent ?objto))
```

```
   :effect (and (in_reach_of_agent ?objto)
            (forall
               (?objfrom - object)
               (when
                  (and
                     (in_reach_of_agent ?objfrom)
                     (not (same_obj ?objfrom ?objto))
                  )
                  (not (in_reach_of_agent ?objfrom))
               )
            )
         )
)

(:action grasp
:parameters (?obj - object ?agent - agent)
:precondition (and (not (holding ?obj))
               (not (handsfull ?agent))
               (in_reach_of_agent ?obj)
               (not (exists (?obj2 - object) (and (inside ?obj ?
                   obj2) (not (open ?obj2)))))
            )
:effect (and (holding ?obj)
            (handsfull ?agent)
            ;; Conditional effects for all predicates involving ?
                obj and ?other_obj
            (forall (?other_obj - object)
               (and (not (inside ?obj ?other_obj))
                     (not (ontop ?obj ?other_obj))
                     (not (under ?obj ?other_obj))
                     (not (under ?other_obj ?obj))
                     (not (nextto ?obj ?other_obj))
                     (not (nextto ?other_obj ?obj))
                     (not (onfloor ?obj ?other_obj))
                     ;; Add other predicates as needed
               )
            )
         )
)

(:action release
   :parameters (?obj - object ?agent - agent)
   :precondition (and (holding ?obj))
   :effect (and (not (holding ?obj))
               (not (handsfull ?agent)))
)

(:action place_ontop
   :parameters (?obj_in_hand - object ?obj - object ?agent - agent)
   :precondition (and (holding ?obj_in_hand)
                  (in_reach_of_agent ?obj))
   :effect (and (ontop ?obj_in_hand ?obj)
               (not (holding ?obj_in_hand))
               (not (handsfull ?agent)))
)

(:action place_inside
```

```
   :parameters (?obj_in_hand - object ?obj - object ?agent - agent)
   :precondition (and (holding ?obj_in_hand)
                  (in_reach_of_agent ?obj)
                  (open ?obj))
   :effect (and (inside ?obj_in_hand ?obj)
               (not (holding ?obj_in_hand))
               (not (handsfull ?agent)))
)

(:action open
   :parameters (?obj - object ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (not (open ?obj))
                  (not (handsfull ?agent)))
   :effect (open ?obj)
)

(:action close
   :parameters (?obj - object ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (open ?obj)
                  (not (handsfull ?agent)))
   :effect (not (open ?obj))
)

(:action slice
   :parameters (?obj - object ?knife - knife_n_01 ?agent - agent)
   :precondition (and (holding ?knife)
                  (in_reach_of_agent ?obj))
   :effect (sliced ?obj)
)

(:action slice-carvingknife
   :parameters (?obj - object ?knife - carving_knife_n_01 ?board -
       countertop_n_01 ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (holding ?knife)
                  (ontop ?obj ?board)
                  (not (sliced ?obj)))
   :effect (sliced ?obj)
)


(:action place_onfloor
   :parameters (?obj_in_hand - object ?floor - floor_n_01 ?agent -
       agent)
   :precondition (and (holding ?obj_in_hand)
                  (in_reach_of_agent ?floor))
   :effect (and (onfloor ?obj_in_hand ?floor)
               (not (holding ?obj_in_hand))
               (not (handsfull ?agent)))
)

(:action place_nextto
   :parameters (?obj_in_hand - object ?obj - object ?agent - agent)
   :precondition (and (holding ?obj_in_hand)
                  (in_reach_of_agent ?obj))
```

```
   :effect (and (nextto ?obj_in_hand ?obj)
                (nextto ?obj ?obj_in_hand)
                (not (holding ?obj_in_hand))
                (not (handsfull ?agent)))
)

(:action place_nextto_ontop
   :parameters (?obj_in_hand - object ?obj1 - object ?obj2 - object
        ?agent - agent)
   :precondition (and (holding ?obj_in_hand)
                   (in_reach_of_agent ?obj1))
   :effect (and (nextto ?obj_in_hand ?obj1)
                (nextto ?obj1 ?obj_in_hand)
                (ontop ?obj_in_hand ?obj2)
                (not (holding ?obj_in_hand))
                (not (handsfull ?agent)))
)

(:action clean_stained_brush
   :parameters (?scrub_brush - scrub_brush_n_01 ?obj - object ?
       agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                   (stained ?obj)
                   (soaked ?scrub_brush)
                   (holding ?scrub_brush))
   :effect (not (stained ?obj))
)

(:action clean_stained_cloth
   :parameters (?rag - piece_of_cloth_n_01 ?obj - object ?agent -
       agent)
   :precondition (and (in_reach_of_agent ?obj)
                   (stained ?obj)
                   (soaked ?rag)
                   (holding ?rag))
   :effect (not (stained ?obj))
)

(:action clean_stained_handowel
   :parameters (?hand_towel - hand_towel_n_01 ?obj - object ?agent -
        agent)
   :precondition (and (in_reach_of_agent ?obj)
                   (stained ?obj)
                   (soaked ?hand_towel)
                   (holding ?hand_towel))
   :effect (not (stained ?obj))
)

(:action clean_stained_towel
   :parameters (?hand_towel - towel_n_01 ?obj - object ?agent -
       agent)
   :precondition (and (in_reach_of_agent ?obj)
                   (stained ?obj)
                   (soaked ?hand_towel)
                   (holding ?hand_towel))
   :effect (not (stained ?obj))
)
```

```
(:action clean_stained_dishtowel
   :parameters (?hand_towel - dishtowel_n_01 ?obj - object ?agent -
       agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (stained ?obj)
                  (soaked ?hand_towel)
                  (holding ?hand_towel))
   :effect (not (stained ?obj))
)

(:action clean_stained_dishwasher
   :parameters (?dishwasher - dishwasher_n_01 ?obj - object ?agent -
       agent)
   :precondition (and (holding ?obj)
                  (in_reach_of_agent ?dishwasher))
   :effect (not (stained ?obj))
)

(:action clean_stained_rag
   :parameters (?rag - rag_n_01 ?obj - object ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (stained ?obj)
                  (soaked ?rag)
                  (holding ?rag))
   :effect (not (stained ?obj))
)

(:action soak
   :parameters (?obj1 - object ?sink - sink_n_01 ?agent - agent)
   :precondition (and (holding ?obj1)
                  (in_reach_of_agent ?sink)
                  (toggled_on ?sink))
   :effect (soaked ?obj1)
)

(:action soak_teapot
   :parameters (?obj1 - object ?agent - agent ?teapot - teapot_n_01)

   :precondition (and (holding ?obj1)
                  (in_reach_of_agent ?teapot))
   :effect (soaked ?obj1)
)

(:action place_under ; place object 1 under object 2
   :parameters (?obj_in_hand - object ?obj - object ?agent - agent)
   :precondition (and (holding ?obj_in_hand)
                  (in_reach_of_agent ?obj))
   :effect (and (under ?obj_in_hand ?obj)
                (not (holding ?obj_in_hand))
                (not (handsfull ?agent)))
)

(:action toggle_on
   :parameters (?obj - object ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (not (handsfull ?agent)))
```

```
   :effect (toggled_on ?obj)
)

(:action clean_dusty_rag
   :parameters (?rag - rag_n_01 ?obj - object ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (dusty ?obj)
                  (holding ?rag))
   :effect (not (dusty ?obj))
)

(:action clean_dusty_towel
   :parameters (?towel - towel_n_01 ?obj - object ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (dusty ?obj)
                  (holding ?towel))
   :effect (not (dusty ?obj))
)

(:action clean_dusty_cloth
   :parameters (?rag - piece_of_cloth_n_01 ?obj - object ?agent -
       agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (dusty ?obj)
                  (holding ?rag))
   :effect (not (dusty ?obj))
)

(:action clean_dusty_brush
   :parameters (?scrub_brush - scrub_brush_n_01 ?obj - object ?
       agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (dusty ?obj)
                  (holding ?scrub_brush))
   :effect (not (dusty ?obj))
)

(:action clean_dusty_vacuum
   :parameters (?vacuum - vacuum_n_04 ?obj - object ?agent - agent)
   :precondition (and (in_reach_of_agent ?obj)
                  (dusty ?obj)
                  (holding ?vacuum))
   :effect (not (dusty ?obj))
)

(:action freeze
   :parameters (?obj - object ?fridge - electric_refrigerator_n_01)
   :precondition (and (inside ?obj ?fridge)
                  (not (frozen ?obj)))
   :effect (frozen ?obj)
)

(:action cook
   :parameters (?obj - object ?pan - pan_n_01)
   :precondition (and (ontop ?obj ?pan)
                  (not (cooked ?obj)))
   :effect (cooked ?obj)
```

```
)

OUTPUT FORMAT:
Concatenate all actions PDDL string into a single string. Output in
     json format where key is "output" and value is your output
    string.
Put all PDDL action strings of the final plan into a JSON list of
    strings.
{{
"output": "YOUR OUTPUT STRING",
"action_plan": ["YOUR", "ACTION", "PLAN"]
}}

Here is an example of the input problem file and unfinished action:
Input:
Problem file:
(define (problem cleaning_floor_0)
   (:domain igibson)

   (:objects
      floor_n_01_1 - floor_n_01
      floor_n_01_2 - floor_n_01
      rag_n_01_1 - rag_n_01
      sink_n_01_1 - sink_n_01
      agent_n_01_1 - agent
   )

   (:init
      (dusty floor_n_01_1)
      (stained floor_n_01_2)
      (ontop rag_n_01_1 table_n_02_1)
      (inroom sink_n_01_1 storage_room)
      (onfloor agent_n_01_1 floor_n_01_2)
   )

   (:goal
      (and
      (not (dusty floor_n_01_1))
      (not (stained floor_n_01_2))
      )
   )
)
Action to be finished:
(:action navigate_to_and_grasp
      :parameters (?objto - object ?agent - agent)
      :precondition ()
      :effect ()
   )
(:action clean-stained-floor-rag
  :parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent - agent)
  :precondition ()
  :effect ()
)
(:action clean-dusty-floor-rag
  :parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent - agent)
  :precondition ()
  :effect ()
```

```
)
(:action soak-rag
  :parameters (?rag - rag_n_01 ?sink - sink_n_01 ?agent - agent)
  :precondition ()
  :effect ()
)
(:action reach
  :parameters (?obj - object ?agent - agent)
  :precondition ()
  :effect ()
)
Output:
{{"output":
"(:action navigate_to_and_grasp
      :parameters (?objto - object ?agent - agent)
      :precondition (not (holding ?objto))
      :effect (and (holding ?objto)
                (in_reach_of_agent ?objto)
                (forall
                   (?objfrom - object)
                   (when
                      (and
                        (or
                          (in_reach_of_agent ?objfrom)
                          (holding ?objfrom)
                         )
                        (not (same_obj ?objfrom ?objto))
                       )
                       (and
                        (not (holding ?objfrom))
                        (not (in_reach_of_agent ?objfrom))
                       )
                   )
                )
            )
 )
(:action clean-stained-floor-rag
  :parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent - agent)
  :precondition (and
             (stained ?floor)
             (soaked ?rag)
             (holding ?rag))
  :effect (and
        (not (stained ?floor))
        (in_reach_of_agent ?floor)
      )

(:action clean-dusty-floor-rag
  :parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent - agent)
  :precondition (and
                 (dusty ?obj)
                 (holding ?rag))
  :effect (and
        (not (dusty ?floor))
        (in_reach_of_agent ?floor)
      )
)
```

```
(:action soak-rag
 :parameters (?rag - rag_n_01 ?sink - sink_n_01 ?agent - agent)
 :precondition (and (holding ?rag)
                (in_reach_of_agent ?sink)
                (toggled_on ?sink))
 :effect (soaked ?rag)

(:action reach
 :parameters (?obj - object ?agent - agent)
 :precondition (and (not (in_reach_of_agent ?obj)))
 :effect (and (in_reach_of_agent ?obj)
              (forall
                 (?objfrom - object)
                 (when
                    (and
                       (in_reach_of_agent ?objfrom)
                       (not (same_obj ?objfrom ?objto))
                    )
                    (not (in_reach_of_agent ?objfrom))
                 )
              )
       )
)",
"action_plan": [\"(navigate_to_and_grasp rag_n_01_1 agent_n_01_1)\",
     \"(reach floor_n_01_1 agent_n_01_1)\", \"(clean-dusty-floor-rag
     rag_n_01_1 floor_n_01_1 agent_n_01_1)\", \"(reach sink_n_01_1
    agent_n_01_1)\", \"(soak-rag rag_n_01_1 sink_n_01_1 agent_n_01_1
    )\", \"(reach floor_n_01_2 agent_n_01_1)\", \"(clean-stained-
    floor-rag rag_n_01_1 floor_n_01_2 agent_n_01_1)\"]
}}

Explanation (not part of the output):
We began by analyzing the goal: `(not (dusty floor_n_01_1))` and `(
    not (stained floor_n_01_2))`. The presence of two distinct
    actions, `clean-dusty-floor-rag` and `clean-stained-floor-rag`,
    was the key insight. We inferred this distinction meant cleaning
     a stain requires a wet rag, while cleaning dust requires a dry
    one. We used a simple real-world logic to connect them: we clean
     dust with a dry rag, but we clean a stain with a wet (soaked)
    rag. We used the domain's `(soaked)` predicate to model this.
This led us to define the core cleaning actions:
1. `clean-stained-floor-rag`: Its main precondition is `(soaked ?
    rag)` and its effect is `(not (stained ?floor))`.
2. `clean-dusty-floor-rag`: Its main precondition is `(not (soaked
    ?rag))` and its effect is `(not (dusty ?floor))`.
This created a logical chain. To satisfy the `(soaked ?rag)`
    precondition, we defined the `soak-rag` action. Its primary
    effect is `(soaked ?rag)`, and its preconditions are that the
    agent is `(holding ?rag)` and `(in_reach_of_agent ?sink)`.
To prevent a deadlock, we modify the initial state so the rag must
    start in a known state. We add (not (soaked rag_n_01_1)) to the
    (:init) block.
To satisfy `(holding ?rag)`, we defined `navigate_to_and_grasp`.
    Its effects are to make the agent `(holding ?objto)` and `(
    handsfull ?agent)`, while also deleting the rag's initial `(
    ontop ...)` predicate.
Finally, we needed to manage the agent's location. We defined `
```

```
    reach` as the navigation action. Its effect is `(
    in_reach_of_agent ?obj)`. Critically, we added a `(forall ...)`
    effect to both `reach` and `navigate_to_and_grasp` to remove the
     agent's *previous* `(in_reach_of_agent ?other)` predicate. This
     ensures the agent is only in one location at a time. We also
    added `(not (in_reach_of_agent ?obj))` as a precondition to `
    reach` to prevent redundant moves.

This set of definitions built the 7-step plan:
1. The agent must first `navigate_to_and_grasp` the rag (which is
    initially dry).
2. With a dry rag, the agent
(a) `reach` the dusty floor and satisfy the precondition for `clean-
    dusty-floor-rag`.
(b) Execute `clean-dusty-floor-rag` induce the effect of the first
    goal.
3. The agent must then
(a) `reach` the sink and
(b) `soak-rag` to make it wet.
4. With a wet rag, the agent can
(a) `reach` the stained floor and and satisfy the precondition for `
    clean-dusty-floor-rag`.
(b) Execute `clean-stained-floor-rag` induce the effect of the
    second goal. This final effect allows the agent to thus
    completing the overall goal.



Now it's your turn. Define the preconditions and effects for a set
    of incomplete action skeletons, then output the valid plan to
    achieve the goal. Strictly format your output using the JSON
    from the OUTPUT FORMAT, no prefix, suffix, or explanation is
    needed.

Input:
Problem file:
{problem_file}
Action to be finished:
{action_handler}
Output:
```

D.4.2 VirtualHome

Template 23: VirtualHome – Transition Modeling Original Template

```
The following is predicates defined in this domain file. Pay
    attention to the types for each predicate.
(define (domain virtualhome)
(:requirements :typing)
   ;; types in virtualhome domain
   (:types
      object character ; Define 'object' and 'character' as types
   )

   ;; Predicates defined on this domain. Note the types for each
```

```
       predicate.
   (:predicates
      (closed ?obj - object) ; obj is closed
      (open ?obj - object) ; obj is open
      (on ?obj - object) ; obj is turned on, or it is activated
      (off ?obj - object) ; obj is turned off, or it is deactivated
      (plugged_in ?obj - object) ; obj is plugged in
      (plugged_out ?obj - object) ; obj is unplugged
      (sitting ?char - character) ; char is sitting, and this
         represents a state of a character
      (lying ?char - character) ; char is lying
      (clean ?obj - object) ; obj is clean
      (dirty ?obj - object) ; obj is dirty
      (obj_ontop ?obj1 ?obj2 - object) ; obj1 is on top of obj2
      (ontop ?char - character ?obj - object) ; char is on obj
      (on_char ?obj - object ?char - character) ; obj is on char
      (inside_room ?obj ?room - object) ; obj is inside room
      (obj_inside ?obj1 ?obj2 - object) ; obj1 is inside obj2
      (inside ?char - character ?obj - object) ; char is inside obj
      (obj_next_to ?obj1 ?obj2 - object) ; obj1 is close to or next
          to obj2
      (next_to ?char - character ?obj - object) ; char is close to
         or next to obj
      (between ?obj1 ?obj2 ?obj3 - object) ; obj1 is between obj2
         and obj3
      (facing ?char - character ?obj - object) ; char is facing obj
      (holds_rh ?char - character ?obj - object) ; char is holding
         obj with right hand
      (holds_lh ?char - character ?obj - object) ; char is holding
         obj with left hand
      (grabbable ?obj - object) ; obj can be grabbed
      (cuttable ?obj - object) ; obj can be cut
      (can_open ?obj - object) ; obj can be opened
      (readable ?obj - object) ; obj can be read
      (has_paper ?obj - object) ; obj has paper
      (movable ?obj - object) ; obj is movable
      (pourable ?obj - object) ; obj can be poured from
      (cream ?obj - object) ; obj is cream
      (has_switch ?obj - object) ; obj has a switch
      (lookable ?obj - object) ; obj can be looked at
      (has_plug ?obj - object) ; obj has a plug
      (drinkable ?obj - object) ; obj is drinkable
      (body_part ?obj - object) ; obj is a body part
      (recipient ?obj - object) ; obj is a recipient
      (containers ?obj - object) ; obj is a container
      (cover_object ?obj - object) ; obj is a cover object
      (surfaces ?obj - object) ; obj has surfaces
      (sittable ?obj - object) ; obj can be sat on
      (lieable ?obj - object) ; obj can be lied on
      (person ?obj - object) ; obj is a person
      (hangable ?obj - object) ; obj can be hanged
      (clothes ?obj - object) ; obj is clothes
      (eatable ?obj - object) ; obj is eatable
      )
   ;; Actions to be predicted
)
```

```
Objective: Given the problem file of pddl, which defines objects in
     the task (:objects), initial conditions (:init) and goal
    conditions (:goal), write the body of PDDL actions (:
    precondition and :effect) given specific action names and
    parameters, so that after executing the actions in some order,
    the goal conditions can be reached from initial conditions.

Each PDDL action definition consists of four main components:
    action name, parameters, precondition, and effect. Here is the
    general format to follow:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)

The :parameters is the list of variables on which the action
    operates. It lists variable names and variable types.

The :precondition is a first-order logic sentence specifying
    preconditions for an action. The precondition consists of
    predicates and 4 possible logical operators: or, and, not,
    exists!
1. The precondition should be structured in Disjunctive Normal Form
     (DNF), meaning an OR of ANDs.
2. The not operator should only be used within these conjunctions.
    For example, (or (and (predicate1 ?x) (predicate2 ?y)) (and (
    predicate3 ?x)))
3. Exists operator is followed by two parts, variable and body. It
    follows the format: exists (?x - variable type) (predicate1 ?x),
     which means there exists an object ?x of certain variable type,
     that predicate1 ?x satisfies.

The :effect lists the changes which the action imposes on the
    current state. The precondition consists of predicates and 6
    possible logical operators: or, and, not, exists, when, forall.
1. The effects should generally be several effects connected by AND
     operators.
2. For each effect, if it is a conditional effect, use WHEN to
    check the conditions. The semantics of (when [condition] [effect
    ]) are as follows: If [condition] is true before the action,
    then [effect] occurs afterwards.
3. If it is not a conditional effect, use predicates directly.
4. The NOT operator is used to negate a predicate, signifying that
    the condition will not hold after the action is executed.
5. Forall operator is followed by two parts, variable and body. It
    follows the format: forall (?x - variable type) (predicate1 ?x),
     which means there for all objects ?x of certain variable type,
    that predicate1 ?x satisfies.

6. An example of effect is (and (when (predicate1 ?x) (not (
    predicate2 ?y))) (predicate3 ?x))

Formally, the preconditions and effects are all clauses <Clause>.
<Clause> := (predicate ?x)
<Clause> := (and <Clause1> <Clause2> ...)
<Clause> := (or <Clause1> <Clause2> ...)
```

```
<Clause> := (not <Clause>)
<Clause> := (when <Clause1> <Clause2>)
<Clause> := (exists (?x - object type) <Clause>)
<Clause> := (forall (?x - object type) <Clause>)

In any case, the occurrence of a predicate should agree with its
    declaration in terms of number and types of arguments defined in
     DOMAIN FILE at the beginning.

Here is an example of the input problem file and unfinished action.
     Observe carefully how to think step by step to write the action
     body of hang_up_clothes:
Input:
Problem file:
(define (problem hang-clothes-problem)
  (:domain household)
  (:objects
   character - character
   shirt - object
   hanger - object
  ) ; This section declares the instances needed for the problem:
      character is an instance of a character; shirt is an instance
      of an object classified as clothes; hanger is an object that
      is suitable for hanging clothes.
  (:init
   (clothes shirt)
   (hangable hanger)
   (holds_rh alice shirt)
   (next_to alice hanger)
  ) ; This section declares the initial conditions. (clothes shirt)
      and (hangable hanger) tells the properties of objects; (
      holds_rh alice shirt) indicates that Alice is holding the
      shirt in her right hand; (next_to alice hanger) means Alice is
       next to the hanger, ready to hang the shirt.
  (:goal
   (and
    (ontop shirt hanger)
   )
  ) ; This section declares the goal. (ontop shirt hanger) is the
      goal, where the shirt should end up hanging on the hanger.
)
Action to be finished:
(:action hang_up_clothes
  :parameters (?char - character ?clothes - object ?hang_obj -
      object)
  :precondition ()
  :effect ()
)

Example output:
Given the objects in the problem file, and what typically needs to
    be true to perform an action like hanging up clothes: 1. clothes
     must indeed be a type of clothing. 2. hang_obj should be
    something on which clothes can be hung (hangable). 3. char
    should be holding the clothes, either in the right or left hand.
     4. char needs to be next to the hanging object to hang the
    clothes. Besides, we need to write preconditions in Disjunctive
```

```
    Normal Form.
These insights guide us to write:
:precondition (or
            (and
               (clothes ?clothes) ; the object must be a piece of
                   clothing
               (hangable ?hang_obj) ; the target must be an object
                   suitable for hanging clothes
               (holds_rh ?char ?clothes) ; character is holding
                   clothes in the right hand
               (next_to ?char ?hang_obj) ; character is next to the
                    hanging object
            )
            (and
               (clothes ?clothes) ; the object must be a piece of
                   clothing
               (hangable ?hang_obj) ; the target must be an object
                   suitable for hanging clothes
               (holds_lh ?char ?clothes) ; character is holding
                   clothes in the left hand
               (next_to ?char ?hang_obj) ; character is next to the
                    hanging object
            )
         )
Effects describe how the world state changes due to the action.
    After hanging up clothes, you'd expect: 1. char is no longer
    holding the clothes. 2. clothes is now on the hang_obj.
These expectations convert into effects:
:effect (and
          (when (holds_rh ?char ?clothes)(not (holds_rh ?char ?
              clothes))) ; if clothes are held in the right hand,
              they are no longer held
          (when (holds_lh ?char ?clothes)(not (holds_lh ?char ?
              clothes))) ; if clothes are held in the left hand,
              they are no longer held
          (ontop ?clothes ?hang_obj) ; clothes are now hanging on
              the object
        )

Combining these parts, the complete hang_up_clothes action becomes:
(:action hang_up_clothes
  :parameters (?char - character ?clothes - object ?hang_obj -
      object)
  :precondition (or
               (and
                 (clothes ?clothes)
                 (hangable ?hang_obj)
                 (holds_rh ?char ?clothes)
                 (next_to ?char ?hang_obj)
               )
               (and
                 (clothes ?clothes)
                 (hangable ?hang_obj)
                 (holds_lh ?char ?clothes)
                 (next_to ?char ?hang_obj)
               )
             )
```

```
  :effect (and
           (when (holds_rh ?char ?clothes)(not (holds_rh ?char ?
              clothes)))
           (when (holds_lh ?char ?clothes)(not (holds_lh ?char ?
              clothes)))
           (ontop ?clothes ?hang_obj)
         )
)

Above is a good example of given predicates in domain file, problem
     file, action names and parameters, how to reason step by step
    and write the action body in PDDL. Pay attention to the usage of
     different connectives and their underlying logic.

Here are some other commonly used actions and their PDDL definition:

(:action put_to
   :parameters (?char - character ?obj - object ?dest - object)
   :precondition (or
     (and
        (hold_lh ?obj) ; The character should hold either with left
            hand or right hand
        (next_to ?char ?dest) ; The character should be close to
            destination
     )
     (and
        (hold_rh ?obj) ; The character should hold either with left
            hand or right hand
        (next_to ?char ?dest) ; The character should be close to
            destination
     )
   )
   :effect (obj_ontop ?obj ?dest) ; The object is now on the
       destination
)
This case illustrates the use of OR to include all possible
    preconditions of an action.

(:action pick_and_place
   :parameters (?char - character ?obj - object ?dest - object)
   :precondition (and
      (grabbable ?obj) ; The object must be grabbable
      (next_to ?char ?obj) ; The character must be next to the
          object
      (not (obj_ontop ?obj ?dest)) ; Ensure the object is not
          already on the destination
   )
   :effect (and
      (obj_ontop ?obj ?dest) ; The object is now on the destination
      (next_to ?char ?dest) ; The character is now next to the
          destination
   )
)
This case illustrates a plain case with only AND operator.

(:action bow
   :parameters (?char - character ?target - character)
```

```
   :precondition (and
      (next_to ?char ?target) ; The character must be next to the
         target to perform the bow
   )
   :effect ()
)
This case illustrates the action can have no effect (or no
    precondition.)

hint:
1. Don't enforce the use of WHEN everywhere.

2. You MUST only use predicates and object types exactly as they
    appear in the domain file at the beginning.

3. Use and only use the arguments provided in :parameters for each
    action. Don't propose additional arguments, unless you are using
     exists or forall.

4. It is possible that action has no precondition or effect.

5. The KEY of the task is to ensure after executing your proposed
    actions in some order, the intial state (:init) in problem file
    can reach the goals (:goal)!!! Pay attention to the initial
    state and final goals in problem file.

6. Preconditions and effects are <Clause> defined above. When there
     is only one predicate, do not use logic connectives.

For actions to be finished, write their preconditions and effects,
    and return in standard PDDL format:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)
Concatenate all actions PDDL string into a single string. Output in
     json format where key is "output" and value is your output
    string: {{"output": YOUR OUTPUT STRING}

Input:
{problem_file}
{action_handler}

Output:
```

Template 24: VirtualHome – Transition Modeling Updated Template (v1)

```
Your task is to act as an expert PDDL (Planning Domain Definition
    Language) solver and designer. Your goal is to define the
    preconditions and effects for a set of incomplete action
    skeletons. You must use the domain's predicates to build a
    logical chain that connects the problem's initial state to its
    goal state. Once the actions are defined, you must output a
    valid plan to achieve the goal.
```

```
The general strategy you might use is
1. Analyze the Problem (Parse Input)
2. Work Backward from the Goal (Define Effects). The easiest way to
     start is by looking at the :goal and mapping it to an action.
3. Define Preconditions (The "Why"). Now that you have an effect,
    ask: "What must be true for this action to happen?"
for examples: Obvious Preconditions: the state must be the opposite
     of the effect, the agent must be present, the agent must have
    the tool; Inferential Preconditions (The "Key Logic"): Look at
    the other actions. Why are there two cleaning actions (clean-
    dusty and clean-stained)? Find a predicate that captures this
    difference. The (soaked ?rag) predicate is the key.
4. Recurse on New Subgoals. You just created new subgoals (
    preconditions). Find the action that achieves this subgoal and
    repeat the same process.
5. Handle State Changes and Deletes (Rigor). Actions don't just add
     facts; they also delete them.
6. Assemble the Final Plan. Finally, trace the complete logical
    chain you have built, starting from the :init state. If the plan
     is not reachable, rework the action definitions again.

The following is predicates defined in this domain file. Pay
    attention to the types for each predicate.
(define (domain virtualhome)
(:requirements :typing)
   ;; types in virtualhome domain
   (:types
      object character ; Define 'object' and 'character' as types
   )

   ;; Predicates defined on this domain. Note the types for each
       predicate.
   (:predicates
      (closed ?obj - object) ; obj is closed
      (open ?obj - object) ; obj is open
      (on ?obj - object) ; obj is turned on, or it is activated
      (off ?obj - object) ; obj is turned off, or it is deactivated
      (plugged_in ?obj - object) ; obj is plugged in
      (plugged_out ?obj - object) ; obj is unplugged
      (sitting ?char - character) ; char is sitting, and this
          represents a state of a character
      (lying ?char - character) ; char is lying
      (clean ?obj - object) ; obj is clean
      (dirty ?obj - object) ; obj is dirty
      (obj_ontop ?obj1 ?obj2 - object) ; obj1 is on top of obj2
      (ontop ?char - character ?obj - object) ; char is on obj
      (on_char ?obj - object ?char - character) ; obj is on char
      (inside_room ?obj ?room - object) ; obj is inside room
      (obj_inside ?obj1 ?obj2 - object) ; obj1 is inside obj2
      (inside ?char - character ?obj - object) ; char is inside obj
      (obj_next_to ?obj1 ?obj2 - object) ; obj1 is close to or next
          to obj2
      (next_to ?char - character ?obj - object) ; char is close to
          or next to obj
      (between ?obj1 ?obj2 ?obj3 - object) ; obj1 is between obj2
          and obj3
      (facing ?char - character ?obj - object) ; char is facing obj
```

```
      (holds_rh ?char - character ?obj - object) ; char is holding
         obj with right hand
      (holds_lh ?char - character ?obj - object) ; char is holding
         obj with left hand
      (grabbable ?obj - object) ; obj can be grabbed
      (cuttable ?obj - object) ; obj can be cut
      (can_open ?obj - object) ; obj can be opened
      (readable ?obj - object) ; obj can be read
      (has_paper ?obj - object) ; obj has paper
      (movable ?obj - object) ; obj is movable
      (pourable ?obj - object) ; obj can be poured from
      (cream ?obj - object) ; obj is cream
      (has_switch ?obj - object) ; obj has a switch
      (lookable ?obj - object) ; obj can be looked at
      (has_plug ?obj - object) ; obj has a plug
      (drinkable ?obj - object) ; obj is drinkable
      (body_part ?obj - object) ; obj is a body part
      (recipient ?obj - object) ; obj is a recipient
      (containers ?obj - object) ; obj is a container
      (cover_object ?obj - object) ; obj is a cover object
      (surfaces ?obj - object) ; obj has surfaces
      (sittable ?obj - object) ; obj can be sat on
      (lieable ?obj - object) ; obj can be lied on
      (person ?obj - object) ; obj is a person
      (hangable ?obj - object) ; obj can be hanged
      (clothes ?obj - object) ; obj is clothes
      (eatable ?obj - object) ; obj is eatable
      )
   ;; Actions to be predicted
)
Objective: Given the problem file of pddl, which defines objects in
     the task (:objects), initial conditions (:init) and goal
    conditions (:goal), write the body of PDDL actions (:
    precondition and :effect) given specific action names and
    parameters.
Each PDDL action definition consists of four main components:
    action name, parameters, precondition, and effect. Here is the
    general format to follow:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)
The :parameters is the list of variables on which the action
    operates. It lists variable names and variable types.
The :precondition is a first-order logic sentence specifying
    preconditions for an action. The precondition consists of
    predicates and 3 possible logical operators: or, and, and not.
    The precondition should be structured in Disjunctive Normal Form
     (DNF), meaning an OR of ANDs. The not operator should only be
    used within these conjunctions. For example, (or (and (
    predicate1 ?x) (predicate2 ?y)) (and (predicate3 ?x)))
The :effect lists the changes which the action imposes on the
    current state. The precondition consists of predicates and 3
    possible logical operators: and, not and when. 1. The effects
    should generally be several effects connected by AND operators.
    2. For each effect, if it is a conditional effect, use WHEN to
```

```
    check the conditions. The semantics of (when [condition] [effect
    ]) are as follows: If [condition] is true before the action,
    then [effect] occurs afterwards. 3. If it is not a conditional
    effect, use predicates directly. 4. The NOT operator is used to
    negate a predicate, signifying that the condition will not hold
    after the action is executed. And example of effect is (and (
    when (predicate1 ?x) (not (predicate2 ?y))) (predicate3 ?x))
In any case, the occurrence of a predicate should agree with its
    declaration in terms of number and types of arguments defined in
     DOMAIN FILE at the beginning.

hint:
1. In many cases WHEN is not necessary. Don't enforce the use of
    WHEN
2. You MUST only use predicates and object types exactly as they
    appear in the domain file at the beginning. Now given the input,
     please fill in the action body for each provided actions in
    PDDL format.
For actions to be finished, write their preconditions and effects,
    and return in standard PDDL format:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)

OUTPUT FORMAT:
Concatenate all actions PDDL string into a single string. Output in
     json format where key is "output" and value is your output
    string.
Put all PDDL action strings of the final plan into a JSON list of
    strings.
{{
"output": "YOUR OUTPUT STRING",
"action_plan": ["YOUR", "ACTION", "PLAN"]
}}

Here is an example of the input problem file and unfinished action:
Input:
Problem file:
(define (problem hang-clothes-problem)
  (:domain virtualhome)
  (:objects
   alice - character
   shirt - object
   hanger - object
  )
  (:init
   (clothes shirt)
   (hangable hanger)
   (holds_rh alice shirt)
   (next_to alice hanger)
  )
  (:goal
   (and
    (ontop shirt hanger)
   )
```

```
  )
)
Action to be finished:
(:action hang_up_clothes
  :parameters (?char - character ?clothes - object ?hang_obj -
      object)
  :precondition ()
  :effect ()
)
Output:
{{"output":
"(:action hang_up_clothes
  :parameters (?char - character ?clothes - object ?hang_obj -
      object)
  :precondition (or
               (and
                 (clothes ?clothes)
                 (hangable ?hang_obj)
                 (holds_rh ?char ?clothes)
                 (next_to ?char ?hang_obj)
               )
               (and
                 (clothes ?clothes)
                 (hangable ?hang_obj)
                 (holds_lh ?char ?clothes)
                 (next_to ?char ?hang_obj)
               )
             )
  :effect (and
           (when (holds_rh ?char ?clothes)(not (holds_rh ?char ?
               clothes)))
           (when (holds_lh ?char ?clothes)(not (holds_lh ?char ?
               clothes)))
           (ontop ?clothes ?hang_obj)
         )
)",
"action_plan": [\"(hang_up_clothes alice shirt hanger)\"]
}}

Explanation (not part of the output):
We began by analyzing the goal: (ontop shirt hanger). We
    immediately identified a critical predicate mismatch. The domain
     file defines (ontop ?char ?obj) for a character on an object,
    but (obj_ontop ?obj1 ?obj2) for an object on an object. We
    concluded the goal must be a typo and the true goal was (
    obj_ontop shirt hanger). This corrected predicate, (obj_ontop ?
    clothes ?hang_obj), became the primary effect of the
    hang_up_clothes action.

Our key insight was that the agent could be holding the clothes in
    either hand. The domain provided distinct predicates: (holds_rh
    ...) and (holds_lh ...). This led us to define the action's
    logic to be robust:
Precondition: We used an (or (holds_rh ?char ?clothes) (holds_lh ?
    char ?clothes)) to accept the item from either hand.
Effect: We used conditional effects ((when ...)). This ensures we
    add (not (holds_rh ...)) only if the right hand was used, and (
```

```
    not (holds_lh ...)) only if the left hand was used, preventing
    state errors.

To complete the action, we looked at the problem's :init state to
    find other necessary facts. This gave us the remaining
    preconditions:
(next_to ?char ?hang_obj) (the agent must be at the hanger).
(clothes ?clothes) (the item must be clothes).
(hangable ?hang_obj) (the target must be hangable).

This single, robust action definition led directly to the plan. We
    checked our action's preconditions against the problem's :init
    block. Since all preconditions were met by the initial state, no
     navigation or grasping actions were needed. The entire plan is
    a single step: The agent executes (hang_up_clothes alice shirt
    hanger), which immediately achieves the goal.

Here is an example of the input problem file and unfinished action:
Input:
Problem file:
(define (problem Wash_clothes)
   (:domain virtualhome)
      (:objects
      character - character
      bathroom clothes_pants washing_machine dining_room
          laundry_detergent - object
   )
   (:init
      (clothes clothes_pants)
      (movable clothes_pants)
      (inside_room clothes_pants bathroom)
      (off washing_machine)
      (containers washing_machine)
      (plugged_in washing_machine)
      (has_switch washing_machine)
      (inside_room washing_machine bathroom)
      (movable laundry_detergent)
      (obj_next_to clothes_pants washing_machine)
      (has_plug washing_machine)
      (obj_next_to washing_machine clothes_pants)
      (can_open washing_machine)
      (closed washing_machine)
      (clean washing_machine)
      (grabbable clothes_pants)
      (pourable laundry_detergent)
      (obj_inside clothes_pants washing_machine)
      (hangable clothes_pants)
      (recipient washing_machine)
      (obj_next_to laundry_detergent washing_machine)
      (inside character dining_room)
      (obj_next_to washing_machine laundry_detergent)
      (inside_room laundry_detergent bathroom)
      (grabbable laundry_detergent)
   )
   (:goal
      (and
         (closed washing_machine)
```

```
          (on washing_machine)
          (plugged_in washing_machine)
          (obj_ontop clothes_pants washing_machine)
          (obj_ontop laundry_detergent washing_machine)
       )
   )
)
Action to be finished:
(:action walk_towards
  :parameters (?char - character obj - object)
  :precondition ()
  :effect ()
)
(:action grab
  :parameters (?char - character ?obj - object)
  :precondition ()
  :effect ()
)
(:action open
  :parameters (?char - character ?obj - object)
  :precondition ()
  :effect ()
)
(:action close
  :parameters (?char - character ?obj - object)
  :precondition ()
  :effect ()
)
(:action obj_puton
  :parameters (?char - character ?obj1 - object ?obj2 - object)
  :precondition ()
  :effect ()
)
(:action switchon
  :parameters (?char - character ?obj - object)
  :precondition ()
  :effect ()
)
Output:
{{"output":
"(:action walk_towards
  :parameters (?char - character ?obj - object)
  :precondition (and
            (not (sitting ?char))
            (not (lying ?char))
  )
  :effect (and
        (next_to ?char ?obj)
        (forall (?far_obj - object)
           (when (not (obj_next_to ?far_obj ?obj)) (not (next_to ?
               char ?far_obj)))
        )
        (forall (?close_obj - object)
           (when (obj_next_to ?close_obj ?obj) (next_to ?char ?
               close_obj))
        )
        (forall (?inside_obj - object)
```

```
            (when (obj_inside ?inside_obj ?obj) (next_to ?char ?
                inside_obj))
        )
        (forall (?ontop_obj - object)
            (when (obj_ontop ?ontop_obj ?obj) (next_to ?char ?
                ontop_obj))
        )
        (forall (?under_obj - object)
            (when (obj_ontop ?obj ?under_obj) (next_to ?char ?
                under_obj))
        )
  )
)
(:action grab
   :parameters (?char - character ?obj - object)
   :precondition (and
              (grabbable ?obj)
              (next_to ?char ?obj)
              (not (exists (?obj2 - object) (and (obj_inside ?obj ?
                  obj2) (or (closed ?obj2) (on ?obj2)))))
              (not (and (exists (?obj3 - object) (holds_lh ?char ?
                  obj3)) (exists (?obj4 - object) (holds_rh ?char ?
                  obj4))))
              )
   :effect (and
         (when (exists (?obj3 - object) (holds_lh ?char ?obj3)) (
             holds_rh ?char ?obj))
         (when (exists (?obj4 - object) (holds_rh ?char ?obj4)) (
             holds_lh ?char ?obj))
         (when (and
               (not (exists (?obj3 - object) (holds_lh ?char ?obj3))
                   )
               (not (exists (?obj4 - object) (holds_rh ?char ?obj4))
                   )
            )
            (holds_rh ?char ?obj) ; Default to right hand
         )
         (forall (?obj2 - object)
            (when (obj_inside ?obj ?obj2)
               (not (obj_inside ?obj ?obj2))
            )
         )
   )
)
(:action open
  :parameters (?char - character ?obj - object)
  :precondition (and
              (can_open ?obj)
              (closed ?obj)
              (next_to ?char ?obj)
              (not (on ?obj))
  )
  :effect (and
         (open ?obj)
         (not (closed ?obj))
  )
)
```

```
(:action close
   :parameters (?char - character ?obj - object)
   :precondition (and
               (can_open ?obj)
               (open ?obj)
               (next_to ?char ?obj)
   )
   :effect (and
         (closed ?obj)
         (not (open ?obj))
   )
)
(:action obj_puton
  :parameters (?char - character ?obj1 - object ?obj2 - object)
  :precondition (or
               (and
               (next_to ?char ?obj2)
               (holds_lh ?char ?obj1)
               )
               (and
               (next_to ?char ?obj2)
               (holds_rh ?char ?obj1)
               )
  )
  :effect (and
        (obj_next_to ?obj1 ?obj2)
        (obj_ontop ?obj1 ?obj2)
        (not (holds_lh ?char ?obj1))
        (not (holds_rh ?char ?obj1))
  )
)
(:action switchon
   :parameters (?char - character ?obj - object)
   :precondition (and
               (has_switch ?obj)
               (off ?obj)
               (plugged_in ?obj)
               (next_to ?char ?obj)

   )
   :effect (and
         (on ?obj)
         (not (off ?obj))
   )
)",
"action_plan": [\"(walk_towards character washing_machine)\", \"(
    grab character laundry_detergent)\", \"(open character
    washing_machine)\", \"(obj_puton character laundry_detergent
    washing_machine)\", \"(grab character clothes_pants)\", \"(
    obj_puton character clothes_pants washing_machine)\", \"(close
    character washing_machine)\", \"(switchon character
    washing_machine)\"]
}}

Explanation (not part of the output):
We began by analyzing the goals: `(obj_ontop clothes_pants
    washing_machine)`, `(obj_ontop laundry_detergent washing_machine
```

```
    )`, `(closed washing_machine)`, and `(on washing_machine)`. The
    key insight came from a major conflict in the initial state: the
     `clothes_pants` start `(obj_inside)` the `(closed)` washer but
    must end up `(obj_ontop)` of it.

This conflict defined the core logic for our actions:
1. The `obj_puton` action is the only one that achieves the `(
    obj_ontop ...)` goal. Its main effect is `(obj_ontop ?obj1 ?obj2
    )` and its precondition is that the agent is holding the item (e
    .g., `(holds_rh ?char ?obj1)` or `(holds_lh ...)`).
2. This made the `grab` action essential. To handle the `(
    obj_inside ...)` conflict, we defined `grab` with two specific
    features:
   * Its precondition `(not (exists (?obj2 ... (closed ?obj2))))`
      prevents grabbing from a closed container.
   * Its effect `(forall (?obj2 ...) (when (obj_inside ...) (not (
      obj_inside ...))))` logically removes the item from any
      container it was in.
3. This, in turn, made the `open` action mandatory. Since the `
    washing_machine` starts `(closed)`, we must `open` it to satisfy
     the `grab` precondition for the pants.

We then defined the remaining actions to satisfy the other goals:
* `close`: Since we were forced to `open` the washer, we needed the
     `close` action to satisfy the final `(closed washing_machine)`
    goal.
* `switchon`: This action was defined to change the washer's state
    from its `(off)` initial state to the `(on)` goal state. We
    added `(plugged_in ?obj)` as a precondition, which was already
    met.
* `walk_towards`: The agent starts in a different room (`
    dining_room`) from all the objects. This action is essential to
    satisfy the `(next_to ?char ?obj)` precondition required by all
    other actions. Its complex `(forall ...)` effects model that
    walking to the `washing_machine` also puts the agent `next_to`
    the `laundry_detergent` (which is `obj_next_to` it) and the `
    clothes_pants` (which are `obj_inside` it).

This set of definitions built the 8-step plan:
1. The agent must `walk_towards` the `washing_machine`. This makes
    all other objects reachable.
2. The agent can then `grab` the `laundry_detergent` (which is
    already outside).
3. Then, the agent must `open` the `washing_machine`.
4. The agent can `obj_puton` the detergent (getting the first goal
    item in place).
5. Now that the washer is open, the agent can `grab` the `
    clothes_pants`.
6. The agent can `obj_puton` the pants.
7. Finally, the agent must `close` the `washing_machine` and
8. `switchon` the `washing_machine` to complete all goals.


Now it's your turn. Define the preconditions and effects for a set
    of incomplete action skeletons, then output the valid plan to
    achieve the goal. Strictly format your output using the JSON
    from the OUTPUT FORMAT, no prefix, suffix, or explanation is
```

```
    needed.

Input:
Problem file:
{problem_file}
Action to be finished:
{action_handler}
Output:
```

Template 25: VirtualHome – Transition Modeling Updated Template (v2)

```
Your task is to act as an expert PDDL (Planning Domain Definition
    Language) solver and designer. Your goal is to define the
    preconditions and effects for a set of incomplete action
    skeletons. You must use the domain's predicates to build a
    logical chain that connects the problem's initial state to its
    goal state. Once the actions are defined, you must output a
    valid plan to achieve the goal.
The general strategy you might use is
1. Analyze the Problem (Parse Input)
2. Work Backward from the Goal (Define Effects). The easiest way to
     start is by looking at the :goal and mapping it to an action.
3. Define Preconditions (The "Why"). Now that you have an effect,
    ask: "What must be true for this action to happen?"
for examples: Obvious Preconditions: the state must be the opposite
     of the effect, the agent must be present, the agent must have
    the tool; Inferential Preconditions (The "Key Logic"): Look at
    the other actions. Why are there two cleaning actions (clean-
    dusty and clean-stained)? Find a predicate that captures this
    difference. The (soaked ?rag) predicate is the key.
4. Recurse on New Subgoals. You just created new subgoals (
    preconditions). Find the action that achieves this subgoal and
    repeat the same process.
5. Handle State Changes and Deletes (Rigor). Actions don't just add
     facts; they also delete them.
6. Assemble the Final Plan. Finally, trace the complete logical
    chain you have built, starting from the :init state. If the plan
     is not reachable, rework the action definitions again.

The following is predicates defined in this domain file. Pay
    attention to the types for each predicate.
(define (domain virtualhome)
(:requirements :typing)
   ;; types in virtualhome domain
   (:types
      object character ; Define 'object' and 'character' as types
   )

   ;; Predicates defined on this domain. Note the types for each
       predicate.
   (:predicates
      (closed ?obj - object) ; obj is closed
      (open ?obj - object) ; obj is open
      (on ?obj - object) ; obj is turned on, or it is activated
      (off ?obj - object) ; obj is turned off, or it is deactivated
```

```
      (plugged_in ?obj - object) ; obj is plugged in
      (plugged_out ?obj - object) ; obj is unplugged
      (sitting ?char - character) ; char is sitting, and this
         represents a state of a character
      (lying ?char - character) ; char is lying
      (clean ?obj - object) ; obj is clean
      (dirty ?obj - object) ; obj is dirty
      (obj_ontop ?obj1 ?obj2 - object) ; obj1 is on top of obj2
      (ontop ?char - character ?obj - object) ; char is on obj
      (on_char ?obj - object ?char - character) ; obj is on char
      (inside_room ?obj ?room - object) ; obj is inside room
      (obj_inside ?obj1 ?obj2 - object) ; obj1 is inside obj2
      (inside ?char - character ?obj - object) ; char is inside obj
      (obj_next_to ?obj1 ?obj2 - object) ; obj1 is close to or next
          to obj2
      (next_to ?char - character ?obj - object) ; char is close to
         or next to obj
      (between ?obj1 ?obj2 ?obj3 - object) ; obj1 is between obj2
         and obj3
      (facing ?char - character ?obj - object) ; char is facing obj
      (holds_rh ?char - character ?obj - object) ; char is holding
         obj with right hand
      (holds_lh ?char - character ?obj - object) ; char is holding
         obj with left hand
      (grabbable ?obj - object) ; obj can be grabbed
      (cuttable ?obj - object) ; obj can be cut
      (can_open ?obj - object) ; obj can be opened
      (readable ?obj - object) ; obj can be read
      (has_paper ?obj - object) ; obj has paper
      (movable ?obj - object) ; obj is movable
      (pourable ?obj - object) ; obj can be poured from
      (cream ?obj - object) ; obj is cream
      (has_switch ?obj - object) ; obj has a switch
      (lookable ?obj - object) ; obj can be looked at
      (has_plug ?obj - object) ; obj has a plug
      (drinkable ?obj - object) ; obj is drinkable
      (body_part ?obj - object) ; obj is a body part
      (recipient ?obj - object) ; obj is a recipient
      (containers ?obj - object) ; obj is a container
      (cover_object ?obj - object) ; obj is a cover object
      (surfaces ?obj - object) ; obj has surfaces
      (sittable ?obj - object) ; obj can be sat on
      (lieable ?obj - object) ; obj can be lied on
      (person ?obj - object) ; obj is a person
      (hangable ?obj - object) ; obj can be hanged
      (clothes ?obj - object) ; obj is clothes
      (eatable ?obj - object) ; obj is eatable
      )
   ;; Actions to be predicted
)
Objective: Given the problem file of pddl, which defines objects in
     the task (:objects), initial conditions (:init) and goal
    conditions (:goal), write the body of PDDL actions (:
    precondition and :effect) given specific action names and
    parameters.
Each PDDL action definition consists of four main components:
    action name, parameters, precondition, and effect. Here is the
```

```
    general format to follow:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)
The :parameters is the list of variables on which the action
    operates. It lists variable names and variable types.
The :precondition is a first-order logic sentence specifying
    preconditions for an action. The precondition consists of
    predicates and 3 possible logical operators: or, and, and not.
    The precondition should be structured in Disjunctive Normal Form
     (DNF), meaning an OR of ANDs. The not operator should only be
    used within these conjunctions. For example, (or (and (
    predicate1 ?x) (predicate2 ?y)) (and (predicate3 ?x)))
The :effect lists the changes which the action imposes on the
    current state. The precondition consists of predicates and 3
    possible logical operators: and, not and when. 1. The effects
    should generally be several effects connected by AND operators.
    2. For each effect, if it is a conditional effect, use WHEN to
    check the conditions. The semantics of (when [condition] [effect
    ]) are as follows: If [condition] is true before the action,
    then [effect] occurs afterwards. 3. If it is not a conditional
    effect, use predicates directly. 4. The NOT operator is used to
    negate a predicate, signifying that the condition will not hold
    after the action is executed. And example of effect is (and (
    when (predicate1 ?x) (not (predicate2 ?y))) (predicate3 ?x))
In any case, the occurrence of a predicate should agree with its
    declaration in terms of number and types of arguments defined in
     DOMAIN FILE at the beginning.

hint:
1. In many cases WHEN is not necessary. Don't enforce the use of
    WHEN
2. You MUST only use predicates and object types exactly as they
    appear in the domain file at the beginning. Now given the input,
     please fill in the action body for each provided actions in
    PDDL format.
For actions to be finished, write their preconditions and effects,
    and return in standard PDDL format:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)

OUTPUT FORMAT:
Concatenate all actions PDDL string into a single string. Output in
     json format where key is "output" and value is your output
    string.
Put all PDDL action strings of the final plan into a JSON list of
    strings.
{{
"output": "YOUR OUTPUT STRING",
"action_plan": ["YOUR", "ACTION", "PLAN"]
}}
```

```
Here is an example of the input problem file and unfinished action:
Input:
Problem file:
(define (problem hang-clothes-problem)
  (:domain virtualhome)
  (:objects
   alice - character
   shirt - object
   hanger - object
  )
  (:init
    (clothes shirt)
    (hangable hanger)
    (holds_rh alice shirt)
    (next_to alice hanger)
  )
  (:goal
    (and
      (ontop shirt hanger)
    )
  )
)
Action to be finished:
(:action hang_up_clothes
  :parameters (?char - character ?clothes - object ?hang_obj -
      object)
  :precondition ()
  :effect ()
)
Output:
{{"output":
"(:action hang_up_clothes
  :parameters (?char - character ?clothes - object ?hang_obj -
      object)
  :precondition (or
               (and
                 (clothes ?clothes)
                 (hangable ?hang_obj)
                 (holds_rh ?char ?clothes)
                 (next_to ?char ?hang_obj)
               )
               (and
                 (clothes ?clothes)
                 (hangable ?hang_obj)
                 (holds_lh ?char ?clothes)
                 (next_to ?char ?hang_obj)
               )
             )
  :effect (and
           (when (holds_rh ?char ?clothes)(not (holds_rh ?char ?
               clothes)))
           (when (holds_lh ?char ?clothes)(not (holds_lh ?char ?
               clothes)))
           (ontop ?clothes ?hang_obj)
         )
)",
"action_plan": [\"(hang_up_clothes alice shirt hanger)\"]
```

```
}}

Explanation (not part of the output):
We began by analyzing the goal: (ontop shirt hanger). We
    immediately identified a critical predicate mismatch. The domain
     file defines (ontop ?char ?obj) for a character on an object,
    but (obj_ontop ?obj1 ?obj2) for an object on an object. We
    concluded the goal must be a typo and the true goal was (
    obj_ontop shirt hanger). This corrected predicate, (obj_ontop ?
    clothes ?hang_obj), became the primary effect of the
    hang_up_clothes action.

Our key insight was that the agent could be holding the clothes in
    either hand. The domain provided distinct predicates: (holds_rh
    ...) and (holds_lh ...). This led us to define the action's
    logic to be robust:
Precondition: We used an (or (holds_rh ?char ?clothes) (holds_lh ?
    char ?clothes)) to accept the item from either hand.
Effect: We used conditional effects ((when ...)). This ensures we
    add (not (holds_rh ...)) only if the right hand was used, and (
    not (holds_lh ...)) only if the left hand was used, preventing
    state errors.

To complete the action, we looked at the problem's :init state to
    find other necessary facts. This gave us the remaining
    preconditions:
(next_to ?char ?hang_obj) (the agent must be at the hanger).
(clothes ?clothes) (the item must be clothes).
(hangable ?hang_obj) (the target must be hangable).

This single, robust action definition led directly to the plan. We
    checked our action's preconditions against the problem's :init
    block. Since all preconditions were met by the initial state, no
     navigation or grasping actions were needed. The entire plan is
    a single step: The agent executes (hang_up_clothes alice shirt
    hanger), which immediately achieves the goal.

Here is an example of the input problem file and unfinished action:
Input:
Problem file:
(define (problem Wash_clothes)
   (:domain virtualhome)
      (:objects
      character - character
      bathroom clothes_pants washing_machine dining_room
          laundry_detergent - object
   )
   (:init
      (clothes clothes_pants)
      (movable clothes_pants)
      (inside_room clothes_pants bathroom)
      (off washing_machine)
      (containers washing_machine)
      (plugged_in washing_machine)
      (has_switch washing_machine)
      (inside_room washing_machine bathroom)
      (movable laundry_detergent)
```

```
       (obj_next_to clothes_pants washing_machine)
       (has_plug washing_machine)
       (obj_next_to washing_machine clothes_pants)
       (can_open washing_machine)
       (closed washing_machine)
       (clean washing_machine)
       (grabbable clothes_pants)
       (pourable laundry_detergent)
       (obj_inside clothes_pants washing_machine)
       (hangable clothes_pants)
       (recipient washing_machine)
       (obj_next_to laundry_detergent washing_machine)
       (inside character dining_room)
       (obj_next_to washing_machine laundry_detergent)
       (inside_room laundry_detergent bathroom)
       (grabbable laundry_detergent)
   )
   (:goal
      (and
         (closed washing_machine)
         (on washing_machine)
         (plugged_in washing_machine)
         (obj_ontop clothes_pants washing_machine)
         (obj_ontop laundry_detergent washing_machine)
      )
   )
)
Action to be finished:
(:action walk_towards
  :parameters (?char - character obj - object)
  :precondition ()
  :effect ()
)
(:action grab
  :parameters (?char - character ?obj - object)
  :precondition ()
  :effect ()
)
(:action open
  :parameters (?char - character ?obj - object)
  :precondition ()
  :effect ()
)
(:action close
  :parameters (?char - character ?obj - object)
  :precondition ()
  :effect ()
)
(:action put_on
  :parameters (?char - character ?obj1 - object ?obj2 - object)
  :precondition ()
  :effect ()
)
(:action switch_on
  :parameters (?char - character ?obj - object)
  :precondition ()
  :effect ()
```

```
)
Output:
{{"output":
"(:action walk_towards
   :parameters (?char - character ?obj - object)
   :precondition (and
              (not (sitting ?char))
              (not (lying ?char))
   )
   :effect (and
         (next_to ?char ?obj)
         (forall (?far_obj - object)
            (when (not (obj_next_to ?far_obj ?obj)) (not (next_to ?
               char ?far_obj)))
         )
         (forall (?close_obj - object)
            (when (obj_next_to ?close_obj ?obj) (next_to ?char ?
               close_obj))
         )
         )
)
(:action grab
   :parameters (?char - character ?obj - object)
   :precondition (and
              (grabbable ?obj)
              (next_to ?char ?obj)
              (not (exists (?obj2 - object) (and (obj_inside ?obj ?
                  obj2) (closed ?obj2))))
              (not (and (exists (?obj3 - object) (holds_lh ?char ?
                  obj3)) (exists (?obj4 - object) (holds_rh ?char ?
                  obj4))))
             )
   :effect (and
         (when (exists (?obj3 - object) (holds_lh ?char ?obj3)) (
             holds_rh ?char ?obj))
         (when (exists (?obj4 - object) (holds_rh ?char ?obj4)) (
             holds_lh ?char ?obj))
         (when
            (not (and (exists (?obj3 - object) (holds_lh ?char ?
               obj3)) (exists (?obj4 - object) (holds_rh ?char ?
               obj4))))
            (holds_rh ?char ?obj)
         )
   )
)
(:action open
  :parameters (?char - character ?obj - object)
  :precondition (and
              (can_open ?obj)
              (closed ?obj)
              (next_to ?char ?obj)
              (not (on ?obj))
  )
  :effect (and
        (open ?obj)
        (not (closed ?obj))
  )
```

```
)
(:action close
   :parameters (?char - character ?obj - object)
   :precondition (and
              (can_open ?obj)
              (open ?obj)
              (next_to ?char ?obj)
   )
   :effect (and
         (closed ?obj)
         (not (open ?obj))
   )
)
(:action put_on
   :parameters (?char - character ?obj1 - object ?obj2 - object)
   :precondition (or
                (and
                (next_to ?char ?obj2)
                (holds_lh ?char ?obj1)
                )
                (and
                (next_to ?char ?obj2)
                (holds_rh ?char ?obj1)
                )
   )
   :effect (and
         (obj_next_to ?obj1 ?obj2)
         (obj_ontop ?obj1 ?obj2)
         (not (holds_lh ?char ?obj1))
         (not (holds_rh ?char ?obj1))
   )
)
(:action switch_on
   :parameters (?char - character ?obj - object)
   :precondition (and
              (has_switch ?obj)
              (off ?obj)
              (plugged_in ?obj)
              (next_to ?char ?obj)

   )
   :effect (and
         (on ?obj)
         (not (off ?obj))
   )
)",
"action_plan": [\"(walk_towards character washing_machine)\", \"(
    grab character laundry_detergent)\", \"(open character
    washing_machine)\", \"(put_on character laundry_detergent
    washing_machine)\", \"(grab character clothes_pants)\", \"(
    put_on character clothes_pants washing_machine)\", \"(close
    character washing_machine)\", \"(switch_on character
    washing_machine)\"]
}}

Explanation (not part of the output):
We began by analyzing the goals: `(obj_ontop clothes_pants
```

```
    washing_machine)`, `(obj_ontop laundry_detergent washing_machine
    )`, `(closed washing_machine)`, and `(on washing_machine)`. The
    key insight came from a major conflict in the initial state: the
     `clothes_pants` start `(obj_inside)` the `(closed)` washer but
    must end up `(obj_ontop)` of it.

This conflict defined the core logic for our actions:
1. The `put_on` action is the only one that achieves the `(
    obj_ontop ...)` goal. Its main effect is `(obj_ontop ?obj1 ?obj2
    )` and its precondition is that the agent is holding the item (e
    .g., `(holds_rh ?char ?obj1)` or `(holds_lh ...)`).
2. This made the `grab` action essential. To handle the `(
    obj_inside ...)` conflict, we defined `grab` with two specific
    features:
   * Its precondition `(not (exists (?obj2 ... (closed ?obj2))))`
      prevents grabbing from a closed container.
   * The second precondition `(not (and (exists (?obj3 ... (
      holds_lh ?char ?obj3)) (exists (?obj4 ... (holds_rh ?char ?
      obj4))))))` make sure at least one hand is free for grabbing.
   * Its effect then corresponding occupy the free hand with the
      target object.
3. This, in turn, made the `open` action mandatory. Since the `
    washing_machine` starts `(closed)`, we must `open` it to satisfy
     the `grab` precondition for the pants.

We then defined the remaining actions to satisfy the other goals:
* `close`: Since we were forced to `open` the washer, we needed the
     `close` action to satisfy the final `(closed washing_machine)`
    goal.
* `switch_on`: This action was defined to change the washer's state
     from its `(off)` initial state to the `(on)` goal state. We
    added `(plugged_in ?obj)` as a precondition, which was already
    met.
* `walk_towards`: The agent starts in a different room (`
    dining_room`) from all the objects. This action is essential to
    satisfy the `(next_to ?char ?obj)` precondition required by all
    other actions. Its complex `(forall ...)` effects model that
    walking to the `washing_machine` also puts the agent `next_to`
    the `laundry_detergent` and the `clothes_pants` (which are `
    obj_next_to` it).

This set of definitions built the 8-step plan:
1. The agent must `walk_towards` the `washing_machine`. This makes
    all other objects reachable.
2. The agent can then `grab` the `laundry_detergent` (which is
    already outside).
3. Then, the agent must `open` the `washing_machine`.
4. The agent can `put_on` the detergent (getting the first goal
    item in place).
5. Now that the washer is open, the agent can `grab` the `
    clothes_pants`.
6. The agent can `put_on` the pants.
7. Finally, the agent must `close` the `washing_machine` and
8. `switch_on` the `washing_machine` to complete all goals.

Here are some other commonly used actions and their PDDL
    definitions. Reuse these whenever possible. They are tried and
```

```
    true for the environment. When you are asked to define actions
    not in this set, make sure they are sensible and generalizable
    actions rather than overfitted actions designed specifically as
    a link toward the goals.

(:action walk_towards
   :parameters (?char - character ?obj - object)
   :precondition (and
             (not (sitting ?char))
             (not (lying ?char))
   )
   :effect (and
         (next_to ?char ?obj)
         (forall (?far_obj - object)
            (when (not (obj_next_to ?far_obj ?obj)) (not (next_to ?
               char ?far_obj)))
         )
         (forall (?close_obj - object)
            (when (obj_next_to ?close_obj ?obj) (next_to ?char ?
               close_obj))
         )
         )
)
(:action walk_into
   :parameters (?char - character ?room - object)
   :precondition (and
             (not (sitting ?char))
             (not (lying ?char))
   )
   :effect (and
         (inside ?char ?room)
         (forall (?far_obj - object)
            (when (not (inside_room ?far_obj ?room)) (not (next_to
               ?char ?far_obj)))
         )
   )
)
(:action sit
   :parameters (?char - character ?obj - object)
   :precondition (and
             (next_to ?char ?obj)
             (sittable ?obj)
             (not (sitting ?char))
   )
   :effect (and
         (sitting ?char)
         (ontop ?char ?obj)
   )
)
(:action standup
   :parameters (?char - character)
   :precondition (or
             (sitting ?char)
             (lying ?char)
   )
   :effect (and
         (not (sitting ?char))
```

```
          (not (lying ?char))
   )
)
(:action grab
   :parameters (?char - character ?obj - object)
   :precondition (and
              (grabbable ?obj)
              (next_to ?char ?obj)
              (not (exists (?obj2 - object) (and (obj_inside ?obj ?
                  obj2) (closed ?obj2))))
              (not (and (exists (?obj3 - object) (holds_lh ?char ?
                  obj3)) (exists (?obj4 - object) (holds_rh ?char ?
                  obj4))))
              )
   :effect (and
          (when (exists (?obj3 - object) (holds_lh ?char ?obj3)) (
              holds_rh ?char ?obj))
          (when (exists (?obj4 - object) (holds_rh ?char ?obj4)) (
              holds_lh ?char ?obj))
          (when
             (not (and (exists (?obj3 - object) (holds_lh ?char ?
                 obj3)) (exists (?obj4 - object) (holds_rh ?char ?
                 obj4))))
             (holds_rh ?char ?obj)
          )
   )
)
(:action open
   :parameters (?char - character ?obj - object)
   :precondition (and
              (can_open ?obj)
              (closed ?obj)
              (next_to ?char ?obj)
              (not (on ?obj))
   )
   :effect (and
          (open ?obj)
          (not (closed ?obj))
   )
)
(:action close
   :parameters (?char - character ?obj - object)
   :precondition (and
              (can_open ?obj)
              (open ?obj)
              (next_to ?char ?obj)
   )
   :effect (and
          (closed ?obj)
          (not (open ?obj))
   )
)
(:action put_on
   :parameters (?char - character ?obj1 - object ?obj2 - object)
   :precondition (or
               (and
               (next_to ?char ?obj2)
```

```
                (holds_lh ?char ?obj1)
                )
                (and
                (next_to ?char ?obj2)
                (holds_rh ?char ?obj1)
                )
    )
    :effect (and
          (obj_next_to ?obj1 ?obj2)
          (obj_ontop ?obj1 ?obj2)
          (not (holds_lh ?char ?obj1))
          (not (holds_rh ?char ?obj1))
    )
)
(:action put_on_character
    :parameters (?char - character ?obj - object)
    :precondition (or
                (holds_lh ?char ?obj)
                (holds_rh ?char ?obj)
             )
    :effect (and
          (on_char ?obj ?char)
          (not (holds_lh ?char ?obj))
          (not (holds_rh ?char ?obj))
    )
)
(:action put_inside
    :parameters (?char - character ?obj1 - object ?obj2 - object)
    :precondition (or
                (and
                (next_to ?char ?obj2)
                (holds_lh ?char ?obj1)
                (not (can_open ?obj2))
                )
                (and
                (next_to ?char ?obj2)
                (holds_lh ?char ?obj1)
                (open ?obj2)
                )
                (and
                (next_to ?char ?obj2)
                (holds_rh ?char ?obj1)
                (not (can_open ?obj2))
                )
                (and
                (next_to ?char ?obj2)
                (holds_rh ?char ?obj1)
                (open ?obj2)
                )
    )
    :effect (and
          (obj_inside ?obj1 ?obj2)
          (not (holds_lh ?char ?obj1))
          (not (holds_rh ?char ?obj1))
    )
)
(:action switch_on
```

```
   :parameters (?char - character ?obj - object)
   :precondition (and
              (has_switch ?obj)
              (off ?obj)
              (plugged_in ?obj)
              (next_to ?char ?obj)

   )
   :effect (and
         (on ?obj)
         (not (off ?obj))
   )
)
(:action switch_off
   :parameters (?char - character ?obj - object)
   :precondition (and
              (has_switch ?obj)
              (on ?obj)
              (next_to ?char ?obj)
   )
   :effect (and
         (off ?obj)
         (not (on ?obj))
   )
)
(:action turn_to
   :parameters (?char - character ?obj - object)
   :precondition ()
   :effect (facing ?char ?obj)
)
(:action wipe
   :parameters (?char - character ?obj1 - object ?obj2 - object)
   :precondition (or
              (and
              (next_to ?char ?obj1)
              (holds_lh ?char ?obj2)
              )
              (and
              (next_to ?char ?obj1)
              (holds_rh ?char ?obj2)
              )
   )
   :effect (and
         (clean ?obj1)
         (not (dirty ?obj1))
   )
)
(:action drop
   :parameters (?char - character ?obj - object ?room - object)
   :precondition (or
              (and
              (holds_lh ?char ?obj)
              (obj_inside ?obj ?room)
              )
              (and
              (holds_rh ?char ?obj)
              (obj_inside ?obj ?room)
```

```
            )
   )
   :effect (and
         (not (holds_lh ?char ?obj))
         (not (holds_rh ?char ?obj))
   )
)
(:action lie
   :parameters (?char - character ?obj - object)
   :precondition (and
            (lieable ?obj)
            (next_to ?char ?obj)
            (not (lying ?char))
   )
   :effect (and
         (lying ?char)
         (ontop ?char ?obj)
         (not (sitting ?char))
   )
)
(:action pour
   :parameters (?char - character ?obj1 - object ?obj2 - object)
   :precondition (or
            (and
            (pourable ?obj1)
            (holds_lh ?char ?obj1)
            (recipient ?obj2)
            (next_to ?char ?obj2)
            )
            (and
            (pourable ?obj1)
            (holds_rh ?char ?obj1)
            (recipient ?obj2)
            (next_to ?char ?obj2)
            )
            (and
            (drinkable ?obj1)
            (holds_lh ?char ?obj1)
            (recipient ?obj2)
            (next_to ?char ?obj2)
            )
            (and
            (drinkable ?obj1)
            (holds_rh ?char ?obj1)
            (recipient ?obj2)
            (next_to ?char ?obj2)
            )
   )
   :effect (obj_inside ?obj1 ?obj2)
)
(:action wash
   :parameters (?char - character ?obj - object)
   :precondition (and
            (next_to ?char ?obj)
   )
   :effect (and
         (clean ?obj)
```

```
          (not (dirty ?obj))
   )
)
(:action plug_in
   :parameters (?char - character ?obj - object)
   :precondition (or
                (and
                (next_to ?char ?obj)
                (has_plug ?obj)
                (plugged_out ?obj)
                )
                (and
                (next_to ?char ?obj)
                (has_switch ?obj)
                (plugged_out ?obj)
                )
              )
   :effect (and
          (plugged_in ?obj)
          (not (plugged_out ?obj))
   )
)
(:action plug_out
   :parameters (?char - character ?obj - object)
   :precondition (and
              (next_to ?char ?obj)
              (has_plug ?obj)
              (plugged_in ?obj)
              (not (on ?obj))
   )
   :effect (and
          (plugged_out ?obj)
          (not (plugged_in ?obj))
   )
)

Now it's your turn. Define the preconditions and effects for a set
    of incomplete action skeletons, then output the valid plan to
    achieve the goal. Strictly format your output using the JSON
    from the OUTPUT FORMAT, no prefix, suffix, or explanation is
    needed.

Input:
Problem file:
{problem_file}
Action to be finished:
{action_handler}
Output:
```

Template 26: VirtualHome – Transition Modeling Updated Template (v3)

```
Your task is to act as an expert PDDL (Planning Domain Definition
    Language) solver and designer. Your goal is to define the
    preconditions and effects for a set of incomplete action
    skeletons. You must use the domain's predicates to build a
```

```
    logical chain that connects the problem's initial state to its
    goal state. Once the actions are defined, you must output a
    valid plan to achieve the goal.
The general strategy you might use is
1. Analyze the Problem (Parse Input)
2. Work Backward from the Goal (Define Effects). The easiest way to
     start is by looking at the :goal and mapping it to an action.
3. Define Preconditions (The "Why"). Now that you have an effect,
    ask: "What must be true for this action to happen?"
for examples: Obvious Preconditions: the state must be the opposite
     of the effect, the agent must be present, the agent must have
    the tool; Inferential Preconditions (The "Key Logic"): Look at
    the other actions. Why are there two cleaning actions (clean-
    dusty and clean-stained)? Find a predicate that captures this
    difference. The (soaked ?rag) predicate is the key.
4. Recurse on New Subgoals. You just created new subgoals (
    preconditions). Find the action that achieves this subgoal and
    repeat the same process.
5. Handle State Changes and Deletes (Rigor). Actions don't just add
     facts; they also delete them.
6. Assemble the Final Plan. Finally, trace the complete logical
    chain you have built, starting from the :init state. If the plan
     is not reachable, rework the action definitions again.

The following is predicates defined in this domain file. Pay
    attention to the types for each predicate.
(define (domain virtualhome)
(:requirements :typing)
   ;; types in virtualhome domain
   (:types
      object character ; Define 'object' and 'character' as types
   )

   ;; Predicates defined on this domain. Note the types for each
       predicate.
   (:predicates
      (closed ?obj - object) ; obj is closed
      (open ?obj - object) ; obj is open
      (on ?obj - object) ; obj is turned on, or it is activated
      (off ?obj - object) ; obj is turned off, or it is deactivated
      (plugged_in ?obj - object) ; obj is plugged in
      (plugged_out ?obj - object) ; obj is unplugged
      (sitting ?char - character) ; char is sitting, and this
          represents a state of a character
      (lying ?char - character) ; char is lying
      (clean ?obj - object) ; obj is clean
      (dirty ?obj - object) ; obj is dirty
      (obj_ontop ?obj1 ?obj2 - object) ; obj1 is on top of obj2
      (ontop ?char - character ?obj - object) ; char is on obj
      (on_char ?obj - object ?char - character) ; obj is on char
      (inside_room ?obj ?room - object) ; obj is inside room
      (obj_inside ?obj1 ?obj2 - object) ; obj1 is inside obj2
      (inside ?char - character ?obj - object) ; char is inside obj
      (obj_next_to ?obj1 ?obj2 - object) ; obj1 is close to or next
          to obj2
      (next_to ?char - character ?obj - object) ; char is close to
          or next to obj
```

```
      (between ?obj1 ?obj2 ?obj3 - object) ; obj1 is between obj2
         and obj3
      (facing ?char - character ?obj - object) ; char is facing obj
      (holds_rh ?char - character ?obj - object) ; char is holding
         obj with right hand
      (holds_lh ?char - character ?obj - object) ; char is holding
         obj with left hand
      (grabbable ?obj - object) ; obj can be grabbed
      (cuttable ?obj - object) ; obj can be cut
      (can_open ?obj - object) ; obj can be opened
      (readable ?obj - object) ; obj can be read
      (has_paper ?obj - object) ; obj has paper
      (movable ?obj - object) ; obj is movable
      (pourable ?obj - object) ; obj can be poured from
      (cream ?obj - object) ; obj is cream
      (has_switch ?obj - object) ; obj has a switch
      (lookable ?obj - object) ; obj can be looked at
      (has_plug ?obj - object) ; obj has a plug
      (drinkable ?obj - object) ; obj is drinkable
      (body_part ?obj - object) ; obj is a body part
      (recipient ?obj - object) ; obj is a recipient
      (containers ?obj - object) ; obj is a container
      (cover_object ?obj - object) ; obj is a cover object
      (surfaces ?obj - object) ; obj has surfaces
      (sittable ?obj - object) ; obj can be sat on
      (lieable ?obj - object) ; obj can be lied on
      (person ?obj - object) ; obj is a person
      (hangable ?obj - object) ; obj can be hanged
      (clothes ?obj - object) ; obj is clothes
      (eatable ?obj - object) ; obj is eatable
      )
   ;; Actions to be predicted
)
Objective: Given the problem file of pddl, which defines objects in
     the task (:objects), initial conditions (:init) and goal
    conditions (:goal), write the body of PDDL actions (:
    precondition and :effect) given specific action names and
    parameters.
Each PDDL action definition consists of four main components:
    action name, parameters, precondition, and effect. Here is the
    general format to follow:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)
The :parameters is the list of variables on which the action
    operates. It lists variable names and variable types.
The :precondition is a first-order logic sentence specifying
    preconditions for an action. The precondition consists of
    predicates and 3 possible logical operators: or, and, and not.
    The precondition should be structured in Disjunctive Normal Form
     (DNF), meaning an OR of ANDs. The not operator should only be
    used within these conjunctions. For example, (or (and (
    predicate1 ?x) (predicate2 ?y)) (and (predicate3 ?x)))
The :effect lists the changes which the action imposes on the
    current state. The precondition consists of predicates and 3
```

```
    possible logical operators: and, not and when. 1. The effects
    should generally be several effects connected by AND operators.
    2. For each effect, if it is a conditional effect, use WHEN to
    check the conditions. The semantics of (when [condition] [effect
    ]) are as follows: If [condition] is true before the action,
    then [effect] occurs afterwards. 3. If it is not a conditional
    effect, use predicates directly. 4. The NOT operator is used to
    negate a predicate, signifying that the condition will not hold
    after the action is executed. And example of effect is (and (
    when (predicate1 ?x) (not (predicate2 ?y))) (predicate3 ?x))
In any case, the occurrence of a predicate should agree with its
    declaration in terms of number and types of arguments defined in
     DOMAIN FILE at the beginning.

hint:
1. In many cases WHEN is not necessary. Don't enforce the use of
    WHEN
2. You MUST only use predicates and object types exactly as they
    appear in the domain file at the beginning. Now given the input,
     please fill in the action body for each provided actions in
    PDDL format.
For actions to be finished, write their preconditions and effects,
    and return in standard PDDL format:
(:action [action name]
  :parameters ([action parameters])
  :precondition ([action precondition])
  :effect ([action effect])
)

OUTPUT FORMAT:
Concatenate all actions PDDL string into a single string. Output in
     json format where key is "output" and value is your output
    string.
Put all PDDL action strings of the final plan into a JSON list of
    strings.
{{
"output": "YOUR OUTPUT STRING",
"action_plan": ["YOUR", "ACTION", "PLAN"]
}}

Here is an example of the input problem file and unfinished action:
Input:
Problem file:
(define (problem hang-clothes-problem)
  (:domain virtualhome)
  (:objects
   alice - character
   shirt - object
   hanger - object
  )
  (:init
   (clothes shirt)
   (hangable hanger)
   (holds_rh alice shirt)
   (next_to alice hanger)
  )
  (:goal
```

```
   (and
     (ontop shirt hanger)
   )
  )
)
Action to be finished:
(:action hang_up_clothes
  :parameters (?char - character ?clothes - object ?hang_obj -
      object)
  :precondition ()
  :effect ()
)
Output:
{{"output":
"(:action hang_up_clothes
  :parameters (?char - character ?clothes - object ?hang_obj -
      object)
  :precondition (or
               (and
                 (clothes ?clothes)
                 (hangable ?hang_obj)
                 (holds_rh ?char ?clothes)
                 (next_to ?char ?hang_obj)
               )
               (and
                 (clothes ?clothes)
                 (hangable ?hang_obj)
                 (holds_lh ?char ?clothes)
                 (next_to ?char ?hang_obj)
               )
             )
  :effect (and
           (when (holds_rh ?char ?clothes)(not (holds_rh ?char ?
               clothes)))
           (when (holds_lh ?char ?clothes)(not (holds_lh ?char ?
               clothes)))
           (ontop ?clothes ?hang_obj)
         )
)",
"action_plan": [\"(hang_up_clothes alice shirt hanger)\"]
}}

Explanation (not part of the output):
We began by analyzing the goal: (ontop shirt hanger). We
    immediately identified a critical predicate mismatch. The domain
     file defines (ontop ?char ?obj) for a character on an object,
    but (obj_ontop ?obj1 ?obj2) for an object on an object. We
    concluded the goal must be a typo and the true goal was (
    obj_ontop shirt hanger). This corrected predicate, (obj_ontop ?
    clothes ?hang_obj), became the primary effect of the
    hang_up_clothes action.

Our key insight was that the agent could be holding the clothes in
    either hand. The domain provided distinct predicates: (holds_rh
    ...) and (holds_lh ...). This led us to define the action's
    logic to be robust:
Precondition: We used an (or (holds_rh ?char ?clothes) (holds_lh ?
```

```
    char ?clothes)) to accept the item from either hand.
Effect: We used conditional effects ((when ...)). This ensures we
    add (not (holds_rh ...)) only if the right hand was used, and (
    not (holds_lh ...)) only if the left hand was used, preventing
    state errors.

To complete the action, we looked at the problem's :init state to
    find other necessary facts. This gave us the remaining
    preconditions:
(next_to ?char ?hang_obj) (the agent must be at the hanger).
(clothes ?clothes) (the item must be clothes).
(hangable ?hang_obj) (the target must be hangable).

This single, robust action definition led directly to the plan. We
    checked our action's preconditions against the problem's :init
    block. Since all preconditions were met by the initial state, no
     navigation or grasping actions were needed. The entire plan is
    a single step: The agent executes (hang_up_clothes alice shirt
    hanger), which immediately achieves the goal.

Here is an example of the input problem file and unfinished action:
Input:
Problem file:
(define (problem Wash_clothes)
   (:domain virtualhome)
      (:objects
      character - character
      bathroom clothes_pants washing_machine dining_room
          laundry_detergent - object
   )
   (:init
      (clothes clothes_pants)
      (movable clothes_pants)
      (inside_room clothes_pants bathroom)
      (off washing_machine)
      (containers washing_machine)
      (plugged_in washing_machine)
      (has_switch washing_machine)
      (inside_room washing_machine bathroom)
      (movable laundry_detergent)
      (obj_next_to clothes_pants washing_machine)
      (has_plug washing_machine)
      (obj_next_to washing_machine clothes_pants)
      (can_open washing_machine)
      (closed washing_machine)
      (clean washing_machine)
      (grabbable clothes_pants)
      (pourable laundry_detergent)
      (obj_inside clothes_pants washing_machine)
      (hangable clothes_pants)
      (recipient washing_machine)
      (obj_next_to laundry_detergent washing_machine)
      (inside character dining_room)
      (obj_next_to washing_machine laundry_detergent)
      (inside_room laundry_detergent bathroom)
      (grabbable laundry_detergent)
   )
```

```
   (:goal
      (and
         (closed washing_machine)
         (on washing_machine)
         (plugged_in washing_machine)
         (obj_ontop clothes_pants washing_machine)
         (obj_ontop laundry_detergent washing_machine)
      )
   )
)
Action to be finished:
(:action walk_towards
  :parameters (?char - character obj - object)
  :precondition ()
  :effect ()
)
(:action grab
  :parameters (?char - character ?obj - object)
  :precondition ()
  :effect ()
)
(:action open
  :parameters (?char - character ?obj - object)
  :precondition ()
  :effect ()
)
(:action close
  :parameters (?char - character ?obj - object)
  :precondition ()
  :effect ()
)
(:action put_on
  :parameters (?char - character ?obj1 - object ?obj2 - object)
  :precondition ()
  :effect ()
)
(:action switch_on
  :parameters (?char - character ?obj - object)
  :precondition ()
  :effect ()
)
Output:
{{"output":
"(:action walk_towards
   :parameters (?char - character ?obj - object)
   :precondition (and
             (not (sitting ?char))
             (not (lying ?char))
   )
   :effect (and
         (next_to ?char ?obj)
         (forall (?far_obj - object)
            (when (not (obj_next_to ?far_obj ?obj)) (not (next_to ?
               char ?far_obj)))
         )
         (forall (?close_obj - object)
            (when (obj_next_to ?close_obj ?obj) (next_to ?char ?
```

```
                close_obj))
          )
          )
)
(:action grab
   :parameters (?char - character ?obj - object)
   :precondition (and
              (grabbable ?obj)
              (next_to ?char ?obj)
              (not (exists (?obj2 - object) (and (obj_inside ?obj ?
                  obj2) (closed ?obj2))))
              (not (and (exists (?obj3 - object) (holds_lh ?char ?
                  obj3)) (exists (?obj4 - object) (holds_rh ?char ?
                  obj4))))
              )
   :effect (and
         (when (exists (?obj3 - object) (holds_lh ?char ?obj3)) (
             holds_rh ?char ?obj))
         (when (exists (?obj4 - object) (holds_rh ?char ?obj4)) (
             holds_lh ?char ?obj))
         (when
            (not (and (exists (?obj3 - object) (holds_lh ?char ?
                obj3)) (exists (?obj4 - object) (holds_rh ?char ?
                obj4))))
            (holds_rh ?char ?obj)
         )
   )
)
(:action open
  :parameters (?char - character ?obj - object)
  :precondition (and
              (can_open ?obj)
              (closed ?obj)
              (next_to ?char ?obj)
              (not (on ?obj))
  )
  :effect (and
        (open ?obj)
        (not (closed ?obj))
  )
)
(:action close
   :parameters (?char - character ?obj - object)
   :precondition (and
              (can_open ?obj)
              (open ?obj)
              (next_to ?char ?obj)
   )
   :effect (and
         (closed ?obj)
         (not (open ?obj))
   )
)
(:action put_on
   :parameters (?char - character ?obj1 - object ?obj2 - object)
   :precondition (or
               (and
```

```
                (next_to ?char ?obj2)
                (holds_lh ?char ?obj1)
                )
                (and
                (next_to ?char ?obj2)
                (holds_rh ?char ?obj1)
                )
    )
    :effect (and
          (obj_next_to ?obj1 ?obj2)
          (obj_ontop ?obj1 ?obj2)
          (not (holds_lh ?char ?obj1))
          (not (holds_rh ?char ?obj1))
    )
)
(:action switch_on
    :parameters (?char - character ?obj - object)
    :precondition (and
              (has_switch ?obj)
              (off ?obj)
              (plugged_in ?obj)
              (next_to ?char ?obj)

    )
    :effect (and
          (on ?obj)
          (not (off ?obj))
    )
)",
"action_plan": [\"(walk_towards character washing_machine)\", \"(
    grab character laundry_detergent)\", \"(open character
    washing_machine)\", \"(put_on character laundry_detergent
    washing_machine)\", \"(grab character clothes_pants)\", \"(
    put_on character clothes_pants washing_machine)\", \"(close
    character washing_machine)\", \"(switch_on character
    washing_machine)\"]
}}

Explanation (not part of the output):
We began by analyzing the goals: `(obj_ontop clothes_pants
    washing_machine)`, `(obj_ontop laundry_detergent washing_machine
    )`, `(closed washing_machine)`, and `(on washing_machine)`. The
    key insight came from a major conflict in the initial state: the
     `clothes_pants` start `(obj_inside)` the `(closed)` washer but
    must end up `(obj_ontop)` of it.

This conflict defined the core logic for our actions:
1. The `put_on` action is the only one that achieves the `(
    obj_ontop ...)` goal. Its main effect is `(obj_ontop ?obj1 ?obj2
    )` and its precondition is that the agent is holding the item (e
    .g., `(holds_rh ?char ?obj1)` or `(holds_lh ...)`).
2. This made the `grab` action essential. To handle the `(
    obj_inside ...)` conflict, we defined `grab` with two specific
    features:
   * Its precondition `(not (exists (?obj2 ... (closed ?obj2))))`
       prevents grabbing from a closed container.
   * The second precondition `(not (and (exists (?obj3 ... (
```

```
       holds_lh ?char ?obj3)) (exists (?obj4 ... (holds_rh ?char ?
       obj4))))))` make sure at least one hand is free for grabbing.
   * Its effect then corresponding occupy the free hand with the
       target object.
3. This, in turn, made the `open` action mandatory. Since the `
    washing_machine` starts `(closed)`, we must `open` it to satisfy
     the `grab` precondition for the pants.

We then defined the remaining actions to satisfy the other goals:
* `close`: Since we were forced to `open` the washer, we needed the
     `close` action to satisfy the final `(closed washing_machine)`
    goal.
* `switch_on`: This action was defined to change the washer's state
     from its `(off)` initial state to the `(on)` goal state. We
    added `(plugged_in ?obj)` as a precondition, which was already
    met.
* `walk_towards`: The agent starts in a different room (`
    dining_room`) from all the objects. This action is essential to
    satisfy the `(next_to ?char ?obj)` precondition required by all
    other actions. Its complex `(forall ...)` effects model that
    walking to the `washing_machine` also puts the agent `next_to`
    the `laundry_detergent` and the `clothes_pants` (which are `
    obj_next_to` it).

This set of definitions built the 8-step plan:
1. The agent must `walk_towards` the `washing_machine`. This makes
    all other objects reachable.
2. The agent can then `grab` the `laundry_detergent` (which is
    already outside).
3. Then, the agent must `open` the `washing_machine`.
4. The agent can `put_on` the detergent (getting the first goal
    item in place).
5. Now that the washer is open, the agent can `grab` the `
    clothes_pants`.
6. The agent can `put_on` the pants.
7. Finally, the agent must `close` the `washing_machine` and
8. `switch_on` the `washing_machine` to complete all goals.

Here are some other commonly used actions and their PDDL
    definitions. Reuse these IN FULL whenever you are asked to
    define actions with these names. They are tried and true for the
     environment. When you are asked to define actions not in this
    set, make sure they are sensible and generalizable actions
    rather than overfitted actions designed specifically as a link
    toward the goals. For examples, the action `walk_towards` will
    need to have the full effect with both forall clauses, even if
    they are redundant for the problem asked.

(:action walk_towards
   :parameters (?char - character ?obj - object)
   :precondition (and
             (not (sitting ?char))
             (not (lying ?char))
   )
   :effect (and
         (next_to ?char ?obj)
         (forall (?far_obj - object)
```

```
            (when (not (obj_next_to ?far_obj ?obj)) (not (next_to ?
                char ?far_obj)))
         )
         (forall (?close_obj - object)
            (when (obj_next_to ?close_obj ?obj) (next_to ?char ?
                close_obj))
         )
         )
)
(:action walk_into
   :parameters (?char - character ?room - object)
   :precondition (and
              (not (sitting ?char))
              (not (lying ?char))
   )
   :effect (and
         (inside ?char ?room)
         (forall (?far_obj - object)
            (when (not (inside_room ?far_obj ?room)) (not (next_to
                ?char ?far_obj)))
         )
   )
)
(:action sit
   :parameters (?char - character ?obj - object)
   :precondition (and
              (next_to ?char ?obj)
              (sittable ?obj)
              (not (sitting ?char))
   )
   :effect (and
         (sitting ?char)
         (ontop ?char ?obj)
   )
)
(:action standup
   :parameters (?char - character)
   :precondition (or
              (sitting ?char)
              (lying ?char)
   )
   :effect (and
         (not (sitting ?char))
         (not (lying ?char))
   )
)
(:action grab
   :parameters (?char - character ?obj - object)
   :precondition (and
              (grabbable ?obj)
              (next_to ?char ?obj)
              (not (exists (?obj2 - object) (and (obj_inside ?obj ?
                  obj2) (closed ?obj2))))
              (not (and (exists (?obj3 - object) (holds_lh ?char ?
                  obj3)) (exists (?obj4 - object) (holds_rh ?char ?
                  obj4))))
              )
```

```
   :effect (and
         (when (exists (?obj3 - object) (holds_lh ?char ?obj3)) (
            holds_rh ?char ?obj))
         (when (exists (?obj4 - object) (holds_rh ?char ?obj4)) (
            holds_lh ?char ?obj))
         (when
            (not (and (exists (?obj3 - object) (holds_lh ?char ?
               obj3)) (exists (?obj4 - object) (holds_rh ?char ?
               obj4))))
            (holds_rh ?char ?obj)
         )
   )
)
(:action open
   :parameters (?char - character ?obj - object)
   :precondition (and
             (can_open ?obj)
             (closed ?obj)
             (next_to ?char ?obj)
             (not (on ?obj))
   )
   :effect (and
         (open ?obj)
         (not (closed ?obj))
   )
)
(:action close
   :parameters (?char - character ?obj - object)
   :precondition (and
             (can_open ?obj)
             (open ?obj)
             (next_to ?char ?obj)
   )
   :effect (and
         (closed ?obj)
         (not (open ?obj))
   )
)
(:action put_on
   :parameters (?char - character ?obj1 - object ?obj2 - object)
   :precondition (or
              (and
              (next_to ?char ?obj2)
              (holds_lh ?char ?obj1)
              )
              (and
              (next_to ?char ?obj2)
              (holds_rh ?char ?obj1)
              )
   )
   :effect (and
         (obj_next_to ?obj1 ?obj2)
         (obj_ontop ?obj1 ?obj2)
         (not (holds_lh ?char ?obj1))
         (not (holds_rh ?char ?obj1))
   )
)
```

```
(:action put_on_character
   :parameters (?char - character ?obj - object)
   :precondition (or
               (holds_lh ?char ?obj)
               (holds_rh ?char ?obj)
            )
   :effect (and
         (on_char ?obj ?char)
         (not (holds_lh ?char ?obj))
         (not (holds_rh ?char ?obj))
   )
)
(:action put_inside
   :parameters (?char - character ?obj1 - object ?obj2 - object)
   :precondition (or
               (and
               (next_to ?char ?obj2)
               (holds_lh ?char ?obj1)
               (not (can_open ?obj2))
               )
               (and
               (next_to ?char ?obj2)
               (holds_lh ?char ?obj1)
               (open ?obj2)
               )
               (and
               (next_to ?char ?obj2)
               (holds_rh ?char ?obj1)
               (not (can_open ?obj2))
               )
               (and
               (next_to ?char ?obj2)
               (holds_rh ?char ?obj1)
               (open ?obj2)
               )
   )
   :effect (and
         (obj_inside ?obj1 ?obj2)
         (not (holds_lh ?char ?obj1))
         (not (holds_rh ?char ?obj1))
   )
)
(:action switch_on
   :parameters (?char - character ?obj - object)
   :precondition (and
            (has_switch ?obj)
            (off ?obj)
            (plugged_in ?obj)
            (next_to ?char ?obj)

   )
   :effect (and
         (on ?obj)
         (not (off ?obj))
   )
)
(:action switch_off
```

```
   :parameters (?char - character ?obj - object)
   :precondition (and
              (has_switch ?obj)
              (on ?obj)
              (next_to ?char ?obj)
   )
   :effect (and
         (off ?obj)
         (not (on ?obj))
   )
)
(:action turn_to
   :parameters (?char - character ?obj - object)
   :precondition ()
   :effect (facing ?char ?obj)
)
(:action wipe
   :parameters (?char - character ?obj1 - object ?obj2 - object)
   :precondition (or
              (and
              (next_to ?char ?obj1)
              (holds_lh ?char ?obj2)
              )
              (and
              (next_to ?char ?obj1)
              (holds_rh ?char ?obj2)
              )
   )
   :effect (and
         (clean ?obj1)
         (not (dirty ?obj1))
   )
)
(:action drop
   :parameters (?char - character ?obj - object ?room - object)
   :precondition (or
              (and
              (holds_lh ?char ?obj)
              (obj_inside ?obj ?room)
              )
              (and
              (holds_rh ?char ?obj)
              (obj_inside ?obj ?room)
              )
   )
   :effect (and
         (not (holds_lh ?char ?obj))
         (not (holds_rh ?char ?obj))
   )
)
(:action lie
   :parameters (?char - character ?obj - object)
   :precondition (and
              (lieable ?obj)
              (next_to ?char ?obj)
              (not (lying ?char))
   )
```

```
   :effect (and
         (lying ?char)
         (ontop ?char ?obj)
         (not (sitting ?char))
   )
)
(:action pour
   :parameters (?char - character ?obj1 - object ?obj2 - object)
   :precondition (or
              (and
              (pourable ?obj1)
              (holds_lh ?char ?obj1)
              (recipient ?obj2)
              (next_to ?char ?obj2)
              )
              (and
              (pourable ?obj1)
              (holds_rh ?char ?obj1)
              (recipient ?obj2)
              (next_to ?char ?obj2)
              )
              (and
              (drinkable ?obj1)
              (holds_lh ?char ?obj1)
              (recipient ?obj2)
              (next_to ?char ?obj2)
              )
              (and
              (drinkable ?obj1)
              (holds_rh ?char ?obj1)
              (recipient ?obj2)
              (next_to ?char ?obj2)
              )
   )
   :effect (obj_inside ?obj1 ?obj2)
)
(:action wash
   :parameters (?char - character ?obj - object)
   :precondition (and
              (next_to ?char ?obj)
   )
   :effect (and
         (clean ?obj)
         (not (dirty ?obj))
   )
)
(:action plug_in
   :parameters (?char - character ?obj - object)
   :precondition (or
               (and
               (next_to ?char ?obj)
               (has_plug ?obj)
               (plugged_out ?obj)
               )
               (and
               (next_to ?char ?obj)
               (has_switch ?obj)
```

```
                (plugged_out ?obj)
                )
              )
   :effect (and
         (plugged_in ?obj)
         (not (plugged_out ?obj))
   )
)
(:action plug_out
   :parameters (?char - character ?obj - object)
   :precondition (and
              (next_to ?char ?obj)
              (has_plug ?obj)
              (plugged_in ?obj)
              (not (on ?obj))
   )
   :effect (and
         (plugged_out ?obj)
         (not (plugged_in ?obj))
   )
)

Now it's your turn. Define the preconditions and effects for a set
    of incomplete action skeletons, then output the valid plan to
    achieve the goal. Strictly format your output using the JSON
    from the OUTPUT FORMAT, no prefix, suffix, or explanation is
    needed.

Input:
Problem file:
{problem_file}
Action to be finished:
{action_handler}
Output:
```