VISUALIZING THOUGHT: CONCEPTUAL DIAGRAMS ENABLE ROBUST PLANNING IN LMMS

Anonymous authors

000

001

002

005

006

009

010

011

012

013

014

016

018

019

021

025 026

027 028

029

031

033

034

037

040

041

042

043

044

046

047

048

052

Paper under double-blind review

ABSTRACT

Human reasoning relies on constructing and manipulating mental models—simplified internal representations of situations used to understand and solve problems. Conceptual diagrams (e.g., a sketch drawn to aid reasoning) externalize these mental models, abstracting irrelevant visual details to efficiently capture how entities interact. In contrast, Large Language Models (LLMs) and Large MultiModal Models (LMMs) predominantly reason through text, limiting their effectiveness on complex multi-step tasks. In this paper, we propose Visual Thinking, a generalizable framework that enables LMMs to reason through multiple chains of self-generated conceptual diagrams, significantly enhancing their combinatorial planning capabilities. Our approach requires no human input beyond the natural language description of the task. It integrates textual and diagrammatic reasoning within an optimized Graph-of-Thought inference framework, enhanced by beam search and depth-wise backtracking. Evaluated on multiple challenging PDDL planning domains, our method substantially improves LMM performance (e.g., GPT-40: $35.5\% \rightarrow 90.2\%$ in Blocksworld) and consistently outperforms text-only search-based inference methods. Additionally, on more difficult planning domains with solution depths up to 40, our approach outperforms the o1-preview reasoning model (e.g., 16 percentage points improvement in Floor Tiles). These results highlight the value of conceptual diagrams as a reasoning medium in LMMs.

1 Introduction

Natural language is a powerful medium for communication, enabling humans to effectively share knowledge and ideas [46, 11, 51]. However, language alone is not an optimal medium for reasoning, as it is inherently linear, sequential, and verbose, making it inefficient for representing complex logical and relational structures [33, 25, 52]. Prior evidence 'human thought' is inherently not verbal, sequential, or linear; rather, it is spatial, parallel, and image-like [52]. Humans construct and utilize internal mental models—simplified analogues of real or hypothetical situations [31, 20, 25, 6], and dynamically manipulate them to represent and predict interactions between objects and solve problems. Crucially, mental models are multimodal, integrating both visual and verbal representations to facilitate learning and robust reasoning [38]. Finally, visual representations have always played a central role in human reasoning and communication, from prehistoric cave art, which predates written language [12], to modern textbook diagrams, scientific figures, and blackboard sketches.

Conceptual diagrams are simplified visual representations that use basic shapes (e.g., circles, squares, lines) to capture how entities interact while abstracting away irrelevant details [52, 33]. They externalize internal mental models, reducing cognitive load and enabling rapid perceptual inference and clearer reasoning [25, 20]. Unlike photorealistic images, which capture fine-grained details of how objects appear, conceptual diagrams encode the structural and relational information essential for reasoning, using colors, relative positions and sizes, and annotations [52, 33]. For example, a square in a diagram might represent a complex object such as a car, with its color, relative size, and position visually encoding relationships to other entities while omitting irrelevant appearance details. Thus, conceptual diagrams are an effective reasoning medium complementary to language, overcoming language's limitations in representing relational structure and aligning closely with humans' multimodal reasoning [52, 25, 19].

Modern large language models (LLMs) and large multimodal models (LMMs) [41, 42, 4] have achieved remarkable success on mathematical and scientific benchmarks, including GSM8K [13], MATH [27], and GPQA [65]. Despite these advances, their reasoning remains inconsistent, particularly on multi-step compositional reasoning, long-horizon planning, and tasks requiring backtracking or error correction [19, 55, 15, 9]. These limitations stem partly from LLMs' reliance on language, which is inherently linear and inefficient for representing complex relational structures [19, 55, 8]. Moreover, the autoregressive architecture of current models enforces sequential next-token prediction, making backtracking challenging [19]. Thus, enabling LMMs to reason with conceptual diagrams and backtrack within a graph-based inference framework offers a promising approach to overcome these bottlenecks.

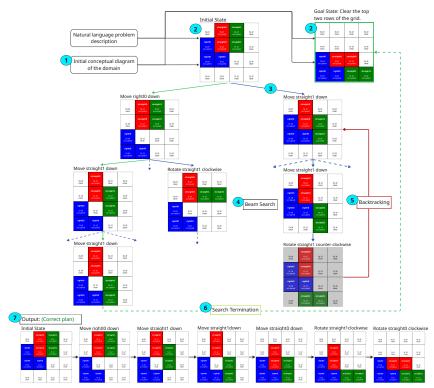


Figure 1: Our proposed approach. Example diagrams are from the Tetris domain, where tiles are moved on a grid to reach a goal state. (1) The model generates multiple diagram schemas and codes for a random instance; their rendered diagrams are ranked, and the code of the top choice is cached (Fig. 2). (2) Conditioned on this code, diagrams for the initial and goal states are generated (Fig. 3). (4) Beam search ranks all candidates at a depth by proximity to the goal, expanding the top k = 4 (Sec. 3). (5) Depth-wise backtracking is applied when all candidate states at a depth fail validation, returning to the deepest available ancestor. (6) The process stops when the goal is reached or a maximum number of steps is visited. (7) The output is the action sequence (plan) plus textual and diagrammatic representations of intermediate states. 1

In this work, we propose Visualizing Thought, a framework that enables LMMs to solve combinatorial problems through multiple multimodal chains of self-generated conceptual diagrams and textual reasoning. Our approach requires no domain-specific modifications or manual engineering to solve any combinatorial problem expressible in the Planning Domain Definition Language (PDDL) [39], given only a natural language specification of the initial state, goal state, and possible actions. Importantly, our method does not rely on predefined visual templates or geometric priors; instead, it generates conceptual diagrams directly from the textual problem description.

Visualizing Thought decomposes inference into a graph of intermediate reasoning steps, where at each node the model selects the next best state. Each node is multimodal, containing both a textual description of the state and a corresponding conceptual diagram (see Fig. 1). At each step, the LMM (i) generates the next state conditioned on the textual and diagrammatic representations of the states in the action path; (ii) produces a diagram schema—a structured set of statements specifying each object's shape, relative size and location, and status; and (iii) generates Matplotlib code from the schema that renders the state diagram. To ensure inference quality, we incorporate guardrails such as diagram-schema self-reflection checks and local (parent—child) and global (entire path from the initial state) validity checks. To manage the exponential growth of the combinatorial search space [7, 64], we integrate beam search to rank validated candidate states at each inference depth and expand only the top k. We also incorporate depth-wise backtracking, which allows the model to revisit earlier validated nodes if all current candidates fail verification. Together, these components enable more efficient exploration of the search space.

Unlike prior approaches that augment language models with visual representations for compositional reasoning [62, 29]—which typically provide an initial visual template for the model to iteratively update—Visualizing Thought relies solely on textual descriptions. The model autonomously generates conceptual diagrams from scratch for every state, without any human-provided visual examples or cues, mirroring how humans use imagination to construct mental models from language. Moreover, instead of producing a single static visualization [29, 58], our method generates evolving sequences of intermediate diagrams that illustrate how the LMM's 'model' of the problem evolves with each reasoning step.

¹ For Figs. 1 and 4, we adjusted the diagram codes only to increase font sizes for better readability. Figs. 2-3 are unmodified generations. Fig. 1 actions are simplified; full action strings include previous/current cells occupied.

	Single inference			Search-based inference (base LLM = GPT-4o)			
Domain	GPT-4o	o1- mini	o1- preview	GoT	Optimized GoT	RAP ⁽²⁰⁾	Visual Thinking
Blocksworld (simple)	35.5%*	56.6% *	97.8%*	50%	58%	58%	90.2%
Blocksworld (hard)	0%	0.9% *	$23.65\%^*$	8%	48%	4%	78 %
Floor Tiles	0%	6%	20%	0%	4%	_	36%
Parking	2%	8%	40%	14%	28%	_	52%
Tetris	0%	2%	26%	0%	12%	_	38%
Elevator	0%	2%	36%	2%	10%	_	48%
Barman	0%	0%	10%	0%	4%	_	30%

Table 1: Accuracy results across all evaluated domains. Each baseline was evaluated on 50 problem instances per domain, except for GPT-40 + Visual Thinking on Blocksworld (simple), where we evaluated 500 instances (the full PlanBench [54]) and achieved 90.2% (451/500). Baseline results marked with * are taken from [56]. RAP reported 51% accuracy on Blocksworld (simple) using Llama 2 [24].

Evaluations across multiple challenging PDDL planning domains demonstrate that our method substantially enhances LLMs' combinatorial reasoning capabilities (Tab. 1). On the widely studied Blocksworld domain, from PlanBench [54], our approach delivers performance gains of 43, 64, and 55 percentage points using Claude 3.5 Sonnet [5], Llama 4 Maverick [40], and GPT-40, respectively (e.g., GPT-40's accuracy rises from 35.5% to 90.2%). Importantly, we contribute *a new, more difficult planning benchmark* with five additional planning domains—Floor Tiles, Parking, Tetris, Elevator, and Barman—with solution depths designed up to 40. On this benchmark, our method succeeds where base models consistently fail (e.g., 36% vs. 0% in Floor Tiles). Furthermore, Visual Thinking (using GPT-40) outperforms the reasoning model, o1-preview [44], across all new domains (e.g., 10% vs. 30% in Barman). Finally, compared to strong search-based inference methods such as Graph-of-Thought [7] and RAP [24] (a Monte Carlo Tree Search framework that uses an LLM to build world models and generate plans), our method improves accuracy by at least 22 percentage points while also reducing inference cost by over 30% and latency by more than 25%.

Crucially, our ablation study on the Blocksworld (simple) domain shows that it is the representation of relational information in conceptual diagrams, not merely the encoded content, that drives these gains. Replacing rendered diagrams with their underlying Matplotlib code, which contains the same spatial and relational data, caused accuracy to collapse from 90.2% to 24%, below the GPT-40 single-inference baseline (35.5%) (Table 3). This sharp decline shows that the compact, parallel, multi-dimensional (2D, color-encoded) representation of object interdependencies in diagrams, rather than their sequential form in text or their syntactically cluttered code representation, is what enables more effective reasoning.

To summarize, our contributions include: 1) a cognitively inspired reasoning framework, Visualizing Thought, that enables LMMs to reason with conceptual diagrams autonomously generated from textual descriptions, with no manual engineering or visual templates required for new domains, within a structured graph-based inference process; 2) empirical evidence that representation of information, not just the content, is critical for reasoning, as replacing rendered diagrams with their code containing the same data causes performance to collapse; and 3) extensive evaluations on PlanBench and a new benchmark of five long-horizon planning domains, where our method consistently outperforms strong search-based baselines and reasoning models (on new domains), demonstrating that conceptual diagrams enable solving problems beyond the reach of purely textual (single-inference or search-based) approaches.

2 RELATED WORK

Multimodal Chain-of-Thought for Reasoning. Several recent works have explored integrating visual representations into the reasoning processes of LLMs and LMMs. Hu et al. [29] equips LMMs with drawing tools to graph equations or mark photorealistic images, but primarily focuses on single-step or shallow problems. Similarly, Wang et al. [58] generates visual aids for spatial reasoning, providing a single refined visualization per problem. In both works, the generated visualizations are typically an approximate or augmentation of high-fidelity illustrations rather than conceptual diagrams drawn using a model-defined mapping of complex entities to simple shapes and colors. Concurrent work, Wu et al. [62], also generates visual and textual intermediate states but requires conditioning the model on a human-provided initial visual representation that supplements the textual description of the problem. This reliance on externally supplied initial visualizations could limit the method's generality and applicability to unseen domains where no such visual initialization exists.

Our approach differs in several key aspects. First, our method autonomously generates conceptual diagrams directly from textual descriptions, without relying on external visual demonstrations or cues, mirroring human ability to construct mental models from language. Second, rather than producing a single visualization, our approach creates multiple chains of intermediate visual states, enabling

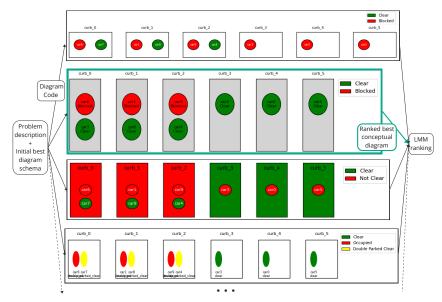


Figure 2: We generate an initial conceptual diagram for each domain by sampling multiple diagram codes. An LMM ranks diagrams based on intuitive and accurate visualization of relational information. The top-ranked diagram's code serves as reference for generating initial and goal state diagrams for all instances. Example shown is the parking domain, where curbs hold up to two cars, and cars can be movable or blocked (double-parked).

parallel multi-hypothesis compositional reasoning through evolving diagrams. Third, our diagrams are conceptual, representing relationships and interactions between entities that are visualized with simple shapes rather than realistic depictions. Finally, our method is applicable to any problem expressible in PDDL format (a general language for planning problems) without domain-specific engineering, given only the textual description of the problem. These distinctions collectively enable a more generalizable and flexible form of diagrammatic reasoning, leading to significant performance gains.

Search and Verification Inference Strategies. Recent methods improve reasoning in LLMs by structuring inference into explicit intermediate steps [60, 32] and employing search-based strategies over multiple reasoning paths [59, 64, 24, 16, 7]. For instance, Yao et al. [64] propose Tree-of-Thought (ToT), which extends CoT to explore a tree of reasoning paths, and Besta et al. [7] introduce Graph-of-Thought (GoT), which structures the reasoning process as a graph, enabling backtracking and aggregation of intermediate reasoning steps. Other works utilize verification and refinement techniques, such as iterative self-reflection and feedback, to enhance reasoning accuracy [37, 45, 50]. Our approach builds upon these methods by integrating a conceptual diagram into each intermediate reasoning step's representation. Moreover, as detailed in Section 3, we extend graph-based inference with beam search, significantly reducing the search space and improving performance on long-horizon planning tasks.

World Modeling with LLMs. Recent research explores planning and reasoning using LLMs by implicitly or explicitly constructing world models from textual descriptions [30, 63, 3, 17, 18, 34, 57]. For instance, Huang et al. [30] show that LLMs implicitly form textual world models for simple planning tasks, while other works explicitly represent states and transitions for structured reasoning [63, 17, 34]. For instance, RAP [24], which we use as a search-based baseline in our evaluation, uses an LLM as both a planning agent and a world modeler, using Monte Carlo Tree Search to simulate future states and rewards that guide its planning. Our work extends these text-based approaches by enabling LMMs to autonomously construct and reason with diagrammatic world models. By defining a visual schema of objects and their statuses using shapes, colors, and spatial arrangements, our method generates diagrams that visually simulate action sequences and state evolutions. This approach provides a more compact representation of relational information, significantly improving performance on complex, multi-step planning tasks, as demonstrated by the substantial accuracy boost on the Blocksworld (hard) domain compared to RAP (78% vs. 4%).

3 METHOD

We propose Visual Thinking, a training-free and model-agnostic framework that integrates textual reasoning with model-generated intermediate conceptual diagrams to enable Large MultiModal Models (LMMs) to solve combinatorial problems, given text-only problem specifications (initial state, goal, admissible actions). Combinatorial problems [61] involve finding a valid sequence of actions from an initial state s_0 to a goal state s_g , given a finite set of possible actions. Our framework, built upon the Graph-of-Thought (GoT) approach [7], decomposes reasoning into discrete nodes in a structured infer-

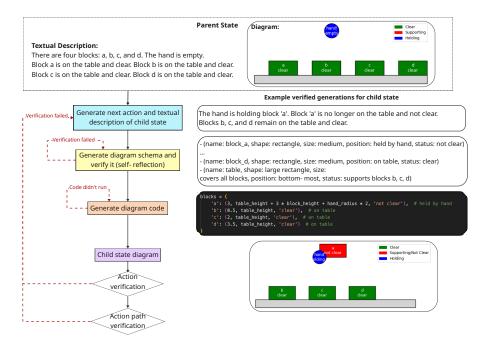


Figure 3: Child state generation pipeline: the LMM selects an action from the parent node, generates a diagram schema and then an executable diagram code, performs self-reflection and verifies that action chosen does not violate any constraints and action path is feasible (example generations are from Blocksworld domain).

ence graph. Through this graph, multiple chains of multimodal states are simultaneously explored toward the goal state. Below, we detail our method following the stages illustrated in Fig. 1. The full implementation, with full prompts and outputs, is available in the supplementary material. An analysis of the prompts and an overview of the code structure are provided in Appendix Sec. B and Sec. C, respectively.

- 1. Generating Initial Conceptual Diagram for Domain. Step 1 in Fig. 1. For each new planning domain, the LMM generates a reference conceptual diagram from a random domain instance. This is done entirely in a zero-shot manner, without manual engineering or external visual cues, conditioned only on the problem's textual description. The process begins with the LMM proposing multiple candidate diagram schemas, which it then verifies by iterating through the objects to confirm the accuracy of their shape, color, status, and relative size and position. The LMM ranks these verified schemas on how clearly they represent object relationships, selecting the top one. Using this schema, the model generates several executable Matplotlib diagram codes. Each rendered diagram is then verified to ensure objects are represented accurately and do not overlap. Finally, the LMM ranks these diagrams based on how effectively they visualize the structure and relationships between objects, and the code for the highest-ranked diagram is cached as a reference for generating the diagram code of the initial and goal states of all instances in the domain. See Fig. 2 for example reference diagrams generated for Parking domain.
- 2. Initial and Goal State Diagram Generation. Step 2 in Fig. 1. We begin the inference process of each instance by generating diagrams for the initial state s_0 and the goal state s_g , conditioned on the domain conceptual diagram code obtained in step 1. When generating the diagram code, the LMM is instructed to adhere to how objects and their statuses are visualized in the reference diagram, while accurately initializing the objects according to the specific instance.
- 3. Intermediate Child State Generation Pipeline. Step 3 in Fig. 1. We denote by s_d an intermediate state at depth d of the graph, represented by a combination of: (i) textual description $T(s_d)$; (ii) a diagram $D(s_d)$; and (iii) the action path $A_{0:d}$ from the initial state s_0 . We iteratively expand the inference graph depth-by-depth in a breadth-first search (BFS) [14] manner, and apply beam search at each depth to select the top-k candidates for further expansion.

From each parent state s_d , we sample $n\!=\!4$ child states. W.l.o.g., next we describe the generation of a single such child state s_{d+1} (Fig. 3). At each node, the LMM first selects the next candidate action a_{d+1} , conditioned on the parent state s_d (which is represented by its diagram, textual description, and action path from initial state). We then generate the textual description $T(s_{d+1})$ of the resulting child state. The candidate action-state pair $(a_{d+1}, T(s_{d+1}))$ is compared to previously generated child states to verify uniqueness. If different, the LMM generates a diagram encoding for s_{d+1} , denoted as $E(s_{d+1})$, which is a structured set of textual statements specifying shapes, sizes, positions, statuses (e.g., colors), and textual identifiers for each object in the state (see Fig. 3 for an example). The $E(s_{d+1})$ undergoes a self-reflection verification to ensure consistency with $T(s_{d+1})$ and the action taken, a_{d+1} , if failed we regenerate it. Subsequently, Matplotlib code $C(s_{d+1})$ is generated

Figure 4: Sequence of intermediate state diagrams in the correct chain, from the initial to the goal state, for one instance across four evaluated domains: Blocksworld, Elevator, Parking, and Floor Tiles (shown top to bottom).

conditioned on $E(s_{d+1})$, $T(s_{d+1})$, and two example diagram codes: the initial state diagram code $C(s_0)$ and the parent state diagram code $C(s_d)$. Code is regenerated if it fails to run.

After generating the child state diagram, we perform two action verifications: (1) a local check that, given the diagram and description of the parent and child state, confirms if the action a_{d+1} complies with domain constraints; and (2) a global check verifying if the entire action path $A_{0:d+1}$ is feasible and efficient for reaching the goal state s_q . If either of these checks fails, the child state is marked as invalid.

- **4. Beam Search**. Step 4 in Fig. 1. Applying GoT [7] to combinatorial problems by naively expanding all nodes (e.g., BFS) results in exponential growth of the search tree [48]. To mitigate this, we use a method inspired by beam search [36], where first all states at each depth d are expanded, generating up to N child states each. The LMM then ranks all candidate child states at depth d+1 based on their proximity to the goal s_g , selecting only the top k=4 states for further expansion. This depth-wise pruning mitigates the exponential growth problem. Moreover, this ranking system resembles human problem-solving strategies, where shallow state-specific heuristics are employed to estimate how close intermediate states are to the goal [22, 10, 47].
- **5. Depth-wise Backtracking**. Step 5 in Fig. 1. Additionally, we implement a depth-wise backtracking mechanism. If all candidate child states at a given depth fail verification, we backtrack to the deepest available ancestor nodes at depth d_{\max} and attempt new expansions. We allow up to B=2 backtracking attempts to any given depth. If all B attempts at depth d_{\max} fail, we mark nodes at that depth as invalid and backtrack further to the next deepest available nodes at depth d_{\max} , where $d_{\max} < d_{\max}$.
- **6. Search Termination**. Step 6 in Fig. 1. The inference process continues iteratively, expanding nodes depth-by-depth, until either the goal state s_g is reached or a predefined computational budget is exhausted. We set two types of computational limits: (1) a maximum number of generated states (120 states for simpler Blocksworld instances, 450 states for more complex domains), and (2) a maximum depth (28 for simpler Blocksworld instances, 100 for other domains). These limits ensure computational efficiency in terms of inference time and API usage costs. If the goal state is not found within these constraints, the search is marked as incomplete. Goal verification occurs at every state expansion, where the LMM compares the diagram and textual description of the current state against those of the goal state.

In summary, our method leverages structured visual reasoning, self-generated conceptual diagrams, and optimized graph-based inference strategies to efficiently solve combinatorial planning problems with an LMM. Each step of our pipeline is visually illustrated in Figures 1, 2, and 3.

4 EVALUATION

To evaluate the proposed approach, we conducted experiments on seven different planning domains, including the popular Blocksworld (simple) [23, 54] and Blocksworld (hard) [56], as well as 5 new domains contributed in this work, prepared using a standard repository of PDDL problem instance generators [1]. Plan correctness was determined with VAL [28]. Experiments were run on a machine with dual-socket Intel Xeon Gold 5220R processors at 2.2 GHz, 35.75 MB L3, 48 cores per node, 8 nodes total.

Baselines. We compare our approach against both single-inference and search-based methods. For single-inference baselines, we evaluated GPT-40, o1-preview [44], and o1-mini [43], each prompted with PDDL instances using templates adapted from [54]. For search-based methods, we evaluate (i) a baseline variant of Graph-of-Thought (GoT), which performs text-only breadth-first search over LLM-generated states; (ii) Optimized GoT, which adds beam search (k = 4) to GoT to enable exploring

Analysis	Corrects (Accuracy)	Incorrects	Incompletes	Avg Depth	Max Depth	Min Depth	Avg Num States
Blocksworld (hard)	78%	10%	12%	20.15	36	18	177.7
Floor Tiles	36%	24%	40%	15.88	25	10	244
Parking	52%	40%	8%	9.6	29	2	149.1
Tetris	38%	60%	2%	6.45	11	4	46.24
Elevator	48%	42%	10%	20.89	26	16	160.30
Barman	30%	40%	30%	24.0	27	22	210.54

Table 2: Analysis of domains. 'Corrects': % instances with correct plans; 'Incorrects': % with incorrect plans; 'Incompletes': % terminated due to state budget; 'Avg/Max/Min Depth': avg/max/min number of actions in correct solutions; 'Avg Num States': average number of states generated across all instances of the domain.

deeper solutions within the compute budget; and (iii) RAP [24], which uses Monte Carlo Tree Search with the LLM generating both the world model and the plan. RAP's compute budget is specified by iteration count; we adopted RAP⁽²⁰⁾, the highest budget reported by the authors in their experiments. RAP was originally evaluated with Llama models; we extended it to GPT-40 for direct comparison. However, RAP's implementation relies on hard-coded prompts and domain-specific parsers available only for Blocksworld, limiting applicability to other domains. Together, these baselines test the limits of purely text-based search and single inference and provide strong points of comparison for our diagram-based framework.

Evaluated Domains. We evaluated our method on combinatorial planning problems [21] from the International Planning Competition (IPC) [2], expressed in PDDL format [39]. These domains include Blocksworld (simple) and Blocksworld (hard) [23, 56], and five additional IPC domains: Floor Tiles [26], Parking, Tetris [53], Elevator, and Barman. Instances for new domains were generated using standard publicly available IPC generators [1]. For Blocksworld (which involves stacking and unstacking blocks), we used 500 simple instances from PlanBench [54] (3–5 blocks) and 50 harder instances (10–20 blocks, following [56]). Floor Tiles features robots painting tiles on a grid; we generated 50 instances with 2–3 rows, 3–5 columns, and 1–2 robots. The Parking domain involves rearranging cars in curbs, with 50 instances using 4-5 curbs and 4-6 cars. The Tetris domain requires rearranging Tetris tiles on a grid, with 50 instances using (4×4) or (6×6) grids. Lastly, the Elevator domain simulates passenger transport in buildings, with 50 instances using 4-5 floors and 10-12 passengers. Figure 4 shows an example sequence of intermediate state diagrams in the correct plan for a subset of domains. Detailed definitions of each domain are provided in Appendix Sec. A.

Translating PDDL to Natural Language and Back. Our method, Visualizing Thought, operates on the natural language description of combinatorial problems. To enable this, we first translate each PDDL domain—the rules and allowed actions—into natural language using a manually engineered five-shot prompt covering five different domains. Each instance, which specifies the initial and goal states, is translated with a one-shot prompt. Our proposed approach then runs entirely on this text representation. After solving the problem, the model's natural language action sequence is translated back into PDDL using a one-shot prompt containing a random (incorrect) plan with correct PDDL syntax. The resulting PDDL plan is then evaluated for correctness using VAL.

4.1 RESULTS AND ANALYSIS

Our main results are presented in Table 1, comparing Visual Thinking (with GPT-4o) against leading reasoning models (01-preview, 01-mini) and strong search-based methods (GoT, Optimized GoT, RAP). Visual Thinking substantially improves over base GPT-4o and consistently outperforms all search-based approaches across domains. On Blocksworld (simple), our method achieves 90.2% accuracy, surpassing GoT, Optimized GoT, and RAP, though slightly trailing o1-preview (97.8%). We conjecture this gap is due to the smaller number of entities, which make the world state easier to track and update in text, and the shallow solution depths, which make these instances easier to solve in a single pass. In contrast, on harder domains, including Blocksworld (hard) and the five new domains, our approach shows considerable, generalizable gains. For example, on Blocksworld (hard) we achieve 78%, compared to 23.65% for o1-preview and 4% for RAP. This trend holds across other domains. Standard GoT often fails completely (e.g., 0% on Floor Tiles and Tetris) due to combinatorial explosion exhausting the budget, and while Optimized GoT mitigates this with beam search, its performance still lags well behind our visual approach. These findings highlight how diagram-based reasoning enables models to capture and analyze complex relational structures more efficiently than purely textual inference.

Table 2 provides further insight into our method's performance on the more challenging domains. Despite significantly deeper solution paths (instances were designed with solution paths of up to 40), our method successfully generates correct plans with as many as 36 sequential actions (Blocksworld (hard)). The primary limitation of our method observed in these experiments is the number of

Ablation	Corrects (Accuracy)	Incorrects	Incompletes	Avg Depth	Max Depth	Avg Num States
GPT-4o + Visual Thinking	90.2%	6.4%	3.4%	10.07	28	38.89
No Diagram (Optimized GoT)	58%	36%	6%	8.28	18	25.86
No Diagram Schema	72%	20%	8%	7.41	18	46.2
No Code Execution	24%	66%	10%	7.82	22	32.7
1-Branching Factor	52%	4%	44%	6.38	16	23.9
2-Branching Factor	70%	6%	24%	8.29	24	26.27
No Backtracking	62%	4%	34%	6.06	14	22.12
No Beam Search	72%	4%	24%	6.61	12	58.31

Table 3: Ablation Study Results. 'Corrects' is our main accuracy metric. See Tab. 2 for columns notation.

incomplete searches (e.g., 40% incomplete in Floor Tiles, 30% in Barman), which arise either when invalid actions are later rejected by local verification using state diagrams, or when inefficient actions that fail to advance toward the goal are pruned by the global check, both leading to exhaustion of the computational budget. We also observe the highest incorrect rate (60%) on the Tetris domain, primarily due to inherently high branching factor in this domain (up to 24 possible actions per state) and complex action parameterization—each action can require up to 7 parameters detailing positions of all sub-tiles, compared to simpler domains like Blocksworld, where actions typically require only 1-2 parameters. Conversely, our largest margin over o1-preview occurs in the Barman domain (30% vs. 10%), likely because diagrammatic representations capture the high number of object statuses and interactions per state in this domain more effectively than text alone.

Model Generalization. To assess the generalizability of our framework, we evaluated it on other state-of-the-art LMMs using 50 instances from the Blocksworld (simple) domain. With Llama 4, our method increased accuracy from a baseline of 10% (single inference) to 74%, an over 7x improvement in accuracy. The improvement was also observed using Claude 3.5 Sonnet, where accuracy increased from 54.8% (using zero shot single inference as reported in [56]) to 98%, achieving state-of-the-art performance on this benchmark. These substantial gains demonstrate that the benefits of our approach are not tied to a specific model architecture but stem from the fundamental advantage of using model-generated conceptual diagrams as a reasoning medium.

Runtime and Cost Analysis. We analyzed the runtime and API costs of our method and all other search-based baselines on 20 instances per domain. On the Blocksworld (simple) domain, our method had a median runtime of 381 (\sim 6 minutes) seconds and a median cost of \$1.04 per instance. For more complex domains, the median runtime was 1038 seconds (\sim 17 minutes) with median cost of \$2.98 per instance. Our approach is significantly more efficient than other search-based methods. On average, it was 31% faster and 36% cheaper than the GoT baseline, and 46% faster and 52% cheaper than RAP²⁰+GPT-40 across all domains, while achieving substantially higher accuracy. Compared to the text-only Optimized GoT, incorporating diagrams added 213 seconds in latency and \$0.71 on average, measured across all domains, but this overhead yielded a 30 percentage point accuracy gain.

4.2 Ablation Studies

To systematically evaluate the contributions of different components of our framework, we conducted ablation studies on 50 instances from the Blocksworld (simple) domain. We examined the impact of state diagrams, diagram schema, diagram code execution, different branching factors, and inference optimizations (beam search and backtracking). Table 3 summarizes the results of these experiments.

Impact of Various Components of State Diagram Generation. We first evaluated the role of state diagrams by removing them entirely from the inference pipeline ("No Diagram"), yielding a text-only optimized Graph-of-Thought approach. This caused accuracy to drop from 90.2% to 58%, underscoring the critical role diagrams play in succinctly representing relational information. Moreover, the average solution depth of correctly solved instances decreased from 10.07 to 8.28, indicating that without diagrams, the model struggled on more complex problems requiring deeper reasoning. In a second experiment, we removed the diagram schema ("No Diagram Schema") from the child-state generation pipeline, instead inferring diagram code directly from textual descriptions of states. Accuracy dropped from 90.2% to 72%, a smaller decline than removing diagrams entirely—showing that diagram schemas further help extract relational information from text, enabling more accurate diagram generation.

Finally, we tested removing the rendered diagrams, providing only the Matplotlib code ("No Code Execution") when generating the next action. This resulted in the most significant performance drop (90.2% to 24%), even below the GPT-40 baseline (35.5%), clearly demonstrating that even though the code encodes the same spatial and relational information as the diagram, the way this information and the interdependencies between objects are represented is crucial for model performance. Using the diagram code directly distracts the model and impairs reasoning, aligning with prior findings that extraneous

details negatively impact model performance [49, 35]. These results reinforce the importance of diagrams as compact, intuitive representations that facilitate rapid perceptual inference and clear relational reasoning [33, 52, 25], and that it is the representation of information in a multi-dimensional (2D, colorencoded) format that significantly aids understanding the interdependencies and reasoning in models.

Impact of Branching Factor. We next investigated the effect of branching factor of the inference graph (the number of candidate child states generated per state) on performance. Reducing the branching factor from 4 (our default) to 2 ("2-Branching Factor") decreased performance from 90.2% to 70%, primarily due to a sharp increase in incomplete searches (24% vs. 3.4%). This suggests that exploring the third or fourth candidate child states, generated at higher temperatures, is frequently necessary to find the correct solution path. Thus, reducing branching factor limits diversity in candidate state generations, leading to more incomplete searches. Additionally, the average depth of correctly solved instances decreased from 10.07 to 8.29, indicating difficulty solving problems with deeper solution depths. Further reducing the branching factor to 1 ("1-Branching Factor"), effectively converting our graph-based inference into a multimodal Chain-of-Thought approach (with diagrams), caused performance to drop even further to 52%. Despite this decline, performance remained well above the GPT-4 baseline (35.5%), underscoring the value of diagrams in improving LMM reasoning even without extensive search.

Impact of Inference Optimizations. Finally, we evaluated the importance of our inference optimizations (backtracking and beam search) on top of the multimodal inference graph. Removing backtracking ("No Backtracking")—yielding a tree-of-thought method with beam search—reduced accuracy from 90.2% to 62%, primarily due to a sharp rise in incomplete searches (34% vs. 3.4%). This occurs because LMM verification steps occasionally produce false negatives, incorrectly invalidating correct states. Without backtracking, the model cannot recover from these errors, leading to incomplete searches as no validated nodes remain at the frontier of the search graph for further expansion.

Similarly, removing beam search ("No Beam Search") lowered accuracy to 72%, with the incomplete search rate increasing to 24%. In this case, incompletes stem from exponential growth in the search space, causing the model to exhaust its computational budget (i.e., the maximum number of states generated) before reaching the goal. Indeed, the average number of generated states increased significantly (58.31 vs. 38.89), underscoring the critical role of beam search in managing combinatorial explosion. Both optimizations are essential for solving deeper combinatorial problems, as shown by the reduced average correct solution depth without them (6.61 without beam search, 6.06 without backtracking, vs. 10.07 with both). These results demonstrate that backtracking and beam search are complementary and crucial for efficient graph-based combinatorial planning.

5 CONCLUSION

Contributions. In this paper, we introduced Visual Thinking, a framework that enables LMMs to solve combinatorial problems by reasoning with conceptual diagrams alongside text. Our contributions are: (i) a cognitively inspired method that autonomously generates conceptual diagrams directly from natural language problem descriptions, requiring no human input for new domains; (ii) a multimodal Graph-of-Thought framework that structures reasoning as sequences of intermediate textual and visual states, integrating beam search and backtracking for efficient long-horizon search; (iii) extensive empirical evidence showing substantial performance gains over single-inference LMMs, specialized reasoning models, and strong search-based baselines across various planning benchmarks; and (iv) ablation results demonstrating that representation of information is critical—reasoning improves when relational information and interdependencies are encoded in diagrams, not merely present in text or code format.

Limitations. As with any search-based inference method, our framework incurs additional computational cost and inference time to explore multiple reasoning trajectories and also to generate visual representations. However, these overheads are manageable in practice; importantly, our approach remains more efficient than prior text-only search methods such as Graph-of-Thought and RAP. Another limitation concerns scope: Visual Thinking was applied to combinatorial planning problems—a core area of computer science with significant real-world applications, such as warehouse optimization, logistics, and scheduling—chosen because their state-based structure makes diagram progression more straightforward. Future work involves extending conceptual diagram generation to more open-ended problems.

Future Work. This work demonstrates a path for LMMs to move beyond purely textual reasoning toward a more powerful, human-like process that integrates visual abstractions. Future work can extend this framework beyond combinatorial planning to more abstract domains, enabling LMMs to produce multimodal outputs such as software architecture diagrams, figures visualizing scientific hypotheses and causal dependencies, or tailored visual aids for educational contexts. Such conceptual diagrams can enhance both model performance and human—AI interaction (e.g., easier verification of code behavior through generated architecture diagrams). This ability to reason and communicate about complex structures multimodally is essential for building AI capable of tackling scientific, creative, and planning tasks.

6 REPRODUCIBILITY STATEMENT

An anonymous supplementary repository includes the full inference code, prompt templates, and scripts used in our experiments, along with the PDDL instances of the new planning benchmark created for the five new domains and sample run results. The end-to-end pipeline is specified in Section 3 and summarized in Figure 1. Dataset sources and instance-generation parameters are detailed in Section 4; formal task and action definitions required to regenerate the PDDL domains are provided in Appendix A; and prompt templates are listed in Appendix B. Experimental setup, baselines, and evaluation protocols are described in Sections 4 and 4.1, with ablations in Section 4.2 and comprehensive results in Tables 1 and 3. Code organization and configuration files are documented in Appendix C.

REFERENCES

- [1] Pddl instance generators. URL http://ipc.icaps-conference.org/domains/.
- [2] Proceedings of the International Planning Competition, 1998–Present. URL http://ipc.icaps-conference.org/. This citation refers to the proceedings and general information about the International Planning Competition (IPC).
- [3] Michael Ahn, Noah Brohan, Noah Brown, Yevgenii Chebotar, Omar Cortes, Byron David, Chelsea Du, Keerthana Duong, Sergey Edunov, Aáron Gomez, et al. Can language models learn from explanations in context? *arXiv preprint arXiv:2204.02329*, 2022.
- [4] Anthropic. Introducing the next generation of claude, 2024. URL https://www.anthropic.com/news/claude-3-family.
- [5] Anthropic. Claude 3.5 sonnet, 2024. URL https://www.anthropic.com/news/claude-3-5-sonnet.
- [6] Peter W Battaglia, Jessica B Hamrick, and Joshua B Tenenbaum. Simulation as an engine of physical scene understanding. *Proceedings of the national academy of sciences*, 110(45): 18327–18332, 2013.
- [7] Maciej Besta, Nils Gerstenberger, Robert Rausch, Tim Fischer, Maximilian Lehmann, Kamil Podstawski, Christoph Huschenbett, Andreas Böttger, and Kristian Kersting. Graph of thoughts: Solving elaborate problems with large language models. 2023.
- [8] Nasim Borazjanizadeh and Steven T Piantadosi. Reliable reasoning beyond natural language. *arXiv preprint arXiv:2407.11373*, 2024.
- [9] Nasim Borazjanizadeh, Roei Herzig, Trevor Darrell, Rogerio Feris, and Leonid Karlinsky. Navigating the labyrinth: Evaluating and enhancing llms' ability to reason about search problems. *arXiv preprint arXiv:2406.12172*, 2024.
- [10] Edward P Chronicle, James N MacGregor, Thomas C Ormerod, and Alistair H Burr. It looks easy! heuristics for combinatorial optimization problems. *The Quarterly Journal of Experimental Psychology*, 59(4):783–800, 2006.
- [11] Herbert H Clark. *Using language*. Cambridge university press, 1996.
- [12] Jean Clottes. *Cave art*. Phaidon Press, 2008.
- [13] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. arXiv preprint arXiv:2110.14168, 2021.
- [14] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 3rd edition, 2009. Breadth-first search algorithm described in Chapter 22.2.
- [15] Antoni Creswell, Lisa Weber, Yogesh Upadhyay, Nicholas Lampson, Jonathan Uesato, Pushmeet Kohli, and Rishabh Das. Selection-inference: Exploiting large language models for interpretable logical reasoning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.
- [16] Antonia Creswell, Murray Shanahan, and Irina Higgins. Selection-inference: Exploiting large language models for interpretable logical reasoning. *arXiv preprint arXiv:2205.09712*, 2022.

- [17] Ishita Dasgupta, Karmanya Murugesan, Orion Jiang, Karthik Murthi, Michael Zhu, Cunjun Yang, Derek Wong, Sriram Banerjee, Antonin Raffin, Andy Zeng, et al. Collaborating with language models for embodied reasoning. *arXiv* preprint arXiv:2207.05608, 2022.
 - [18] Danny Driess, Aakanksha Chowdhery, Josef Schrittwieser, Ferenc Laskin, Michael and, Ayzaan Shah, Carolina Castro, Nadine Clark, Omar Cortés, Fergal Aherne, et al. Palm-e: An embodied multimodal language model. arXiv preprint arXiv:2303.03378, 2023.
 - [19] Nouha Dziri, Siddhartha Chaudhuri, Christopher D Manning, Percy Liang, and Tatsunori Hashimoto. Faith and fate: Limits of transformers on compositionality. In *International Conference on Machine Learning*, pp. 10454–10481. PMLR, 2023.
 - [20] Dedre Gentner and Albert L Stevens. Mental models, psychology of. *International encyclopedia of the social & behavioral sciences*, pp. 9609–9613, 2001.
 - [21] Malik Ghallab, Maria Fox, Derek Long, David Smith, Anthony Cremers, and Jörg Hoffmann. *PDDL2. 2-the language for describing planners*. Yale University Department of Computer Science New Haven, CT, USA, 2004.
 - [22] Gerd Gigerenzer and Daniel G Goldstein. Reasoning the fast and frugal way: Models of bounded rationality. *Psychological review*, 103(4):650, 1996.
 - [23] Neelima Gupta and Dana S Nau. The complexity of blocks-world planning. *AAAI*, pp. 640–646, 1991. Relevant sentence: "The complexity of blocks-world planning is examined in detail.".
 - [24] Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*, 2023.
 - [25] Mary Hegarty. Mechanical reasoning by mental simulation. *Trends in cognitive sciences*, 8(6): 280–285, 2004.
 - [26] Malte Helmert. Landmarks in planning. In *Principles of knowledge representation and reasoning*, pp. 97–106, 2000. Relevant sentence: "Landmarks are propositions that must be true in every plan that solves a given planning problem.".
 - [27] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv* preprint arXiv:2103.03874, 2021.
 - [28] Richard Howey, Derek Long, and Maria Fox. Val: automatic plan validation, continuous effects and mixed initiative planning using pddl. In *Proceedings 16th IEEE International Conference on Tools with Artificial Intelligence*, pp. 294–301. IEEE, 2004.
 - [29] Yushi Hu, Weijia Shi, Xingyu Fu, Dan Roth, Mari Ostendorf, Luke Zettlemoyer, Noah A Smith, and Ranjay Krishna. Visual sketchpad: Sketching as a visual chain of thought for multimodal language models. *arXiv preprint arXiv:2406.09403*, 2024.
 - [30] Wenlong Huang, Fei Xia, Ayzaan Shah, Andy Zeng, Michael Janner, Sergey Levine, Chelsea Finn, Animesh Garg, Anca D Dragan, and Liwei Song. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pp. 8844–8867. PMLR, 2022.
 - [31] Philip Nicholas Johnson-Laird. *Mental models*. Harvard University Press, 1983.
 - [32] Takeshi Kojima, Shixiang Sano, Hideaki Yan, Keizo Furukawa, Jun Sadaie, and Kunihiko Yanai. Large language models are zero-shot reasoners. In *Advances in neural information processing systems*, 2022.
 - [33] Jill H Larkin and Herbert A Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science*, 11(1):65–99, 1987.
 - [34] Keming Liu, Tianhua Zhang, Yujie Chen, Jia Gao, Wei Zhang, Xiang Zhou, Tao Jiang, Junsheng Wang, and Tie-Yan Chen. LLM+: Augmenting language models with explicit state representations. *arXiv preprint arXiv:2305.06975*, 2023.

- [35] Nelson F Liu, Kevin Shen, Keyulu Zhang, and Jian Tang. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*, 2023.
 - [36] Bruce T Lowerre. The harpy speech recognition system. *Ph.D. Dissertation, Carnegie Mellon University*, 1976. Often cited as the origin of Beam Search, though the term "beam search" wasn't explicitly used in this dissertation. Lowerre describes a similar approach for reducing search space in speech recognition.
 - [37] Aman Madaan, Prateek Khapra, Yulia Tsvetkov, and Ruslan Salakhutdinov. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.
 - [38] Richard E Mayer. Multimedia learning. Cambridge university press, 2002.
 - [39] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.
 - [40] Meta. Llama 4, 2025. URL https://ai.meta.com/blog/llama-4-multimodal-intelligence/.
 - [41] OpenAI. Gpt-4 technical report. ArXiv, abs/2303.08774, 2023.
 - [42] OpenAI. Hello gpt-4o, 2024. URL https://openai.com/index/hello-gpt-4o/.
 - [43] OpenAI. Openai ol-mini, 2024. URL https://openai.com/index/openai-ol-mini-advancing-cost-efficient-reasoning/.
 - [44] OpenAI. Introducing openai o1-preview, 2024. URL https://openai.com/index/introducing-openai-o1-preview/.
 - [45] Sreejan Kumar Paul, Michael Laskin, Kevin Chiang, Yi Jiang, Nan Du, Sergey Levine, Roger Grosse, Yuan Yao, et al. Refiner: Reasoning feedback on intermediate representations. *arXiv* preprint arXiv:2304.01904, 2023.
 - [46] Steven Pinker and Paul Bloom. Natural language and natural selection. *Behavioral and Brain Sciences*, 13(4):707–727, 1990.
 - [47] Zygmunt Pizlo and Zheng Li. Solving 756 combinatorial problems: The 15-puzzle. *Memory & cognition*, 33(6):1069–1084, 2005.
 - [48] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson Education, 3rd edition, 2010. Section 10.1.1: Complexity of Classical Planning.
 - [49] Zheng Shi, Yujie Lu, and Albert G Schwing. Large language models can be easily distracted by irrelevant context. *arXiv* preprint arXiv:2302.00093, 2023.
 - [50] Noah Shinn, Beck Labash, Ashwin Gopinath, Rohan Lee, Maarten Park, Eran Davidson, Denny Zhou, Quoc V Le Liang, and Ed Chi. Reflexion: Language agents with verbal reinforcement learning. arXiv preprint arXiv:2303.11366, 2023.
 - [51] Michael Tomasello. *Origins of human communication*. MIT press, 2010.
 - [52] Barbara Tversky. Visualizing thought. *Topics in cognitive science*, 3(1):159–185, 2011.
 - [53] Mauro Vallati. Automated synthesis of tetris domains. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. Relevant sentence: "This paper presents an automated approach to synthesizing planning domain models for Tetris puzzles.".
 - [54] Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. Advances in Neural Information Processing Systems, 36:38975–38987, 2023.
 - [55] Karthik Valmeekam, Raymond Nguyen, Quoc V Le, Lin Li, Nate Kushman, and Oleksandr Polozov. Planning with large language models for code generation. arXiv preprint arXiv:2302.04761, 2023.

- [56] Karthik Valmeekam, Kaya Stechly, Atharva Gundawar, and Subbarao Kambhampati. Planning in strawberry fields: Evaluating and improving the planning and scheduling capabilities of lrm o1. arXiv preprint arXiv:2410.02162, 2024.
 - [57] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. arXiv preprint arXiv:2305.16291, 2023.
 - [58] Pan Wang, Yuchen Li, Jiaqi Liu, Xiaojun Wang, and William Yang Wang. Whiteboard of thought: A framework for llms to plan and reason with visual reminders. arXiv preprint arXiv:2405.19448, 2024.
 - [59] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed Chi, Denny Zhou, et al. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.
 - [60] Jason Wei, Denny Zhou, Quoc V Le, and Qu Zhou. Chain-of-thought prompting elicits reasoning in large language models. In Advances in Neural Information Processing Systems, volume 35, pp. 24824–24840, 2022.
 - [61] Wikipedia. Combinatorial optimization, 2025. URL https://en.wikipedia.org/ wiki/Combinatorial_optimization.
 - [62] Junyi Wu, Yunfan Jiang, Guanzhi Wang, Yuheng Zhao, Ajay Mandlekar, Anima Anandkumar, Linxi Fan, and Yuke Zhu. Imagine while reasoning in space: Multimodal visualization-of-thought. 2024. URL https://arxiv.org/abs/2501.07542.
 - [63] Shunyu Yao, Yuhao Zhao, Yi Geng, Angeliki Lazaridou, Karthik Grosse, Yuan Cao, John Langford, Denny Zhou, Hal Daume III, et al. React: Synergizing reasoning and acting in language models. In *Advances in Neural Information Processing Systems*, 2022.
 - [64] Shunyu Yao, Dian Yu, Jeffrey Zhao, et al. Tree of thought: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 478–494, 2023.
 - [65] Yuning Zhong, Ruixiang Ding, Omid Gheini, Predrag Bosnic, David Dohan, Nadav Sag, Ben Zhang, Lin Wang, Yizhong Yilmaz, et al. Gpqa: A benchmark for evaluating general-purpose question answering capabilities of large language models. *arXiv preprint arXiv:2305.12479*, 2023.

APPENDIX

A DOMAIN DEFINITIONS

This section provides formal definitions for the five planning domains introduced in our benchmark. For each domain, we describe the main objective and provide a detailed specification of the available actions, including their purpose, preconditions, and effects.

A.1 BARMAN DOMAIN

The Barman domain models the task of a bartender preparing cocktails. The agent must use two hands to manipulate containers (shots, shakers), ingredients from dispensers, and mix them to create specific cocktails. The state of each object includes its location (on table or held), contents, cleanliness, and for shakers, fill level and whether it has been shaken.

ACTIONS

- 1. **grasp (hand, container)**: An empty hand picks up a container from the table.
 - **Preconditions**: The container is on the table; the hand is empty.
 - Effects: The container is no longer on the table; the hand now holds the container and is no longer empty.
- 2. **leave (hand, container)**: A hand places a held container onto the table.
 - **Preconditions**: The hand is holding the container.

- Effects: The container is now on the table; the hand becomes empty.
- 3. fill-shot(shot, ingredient, hand1, hand2, dispenser): Fills a clean, empty shot with an ingredient.
 - **Preconditions**: hand1 holds the shot; hand2 is empty; the dispenser provides the ingredient; the shot is empty and clean.
 - Effects: The shot now contains the ingredient and is no longer empty or clean (it becomes used).
- 4. refill-shot(shot, ingredient, hand1, hand2, dispenser): Refills a used shot with the same ingredient it previously held.
 - **Preconditions**: hand1 holds the shot; hand2 is empty; the dispenser provides the ingredient; the shot is empty and was previously used with this ingredient.
 - Effects: The shot now contains the ingredient and is no longer empty.
- 5. **empty-shot (hand, shot, beverage)**: Empties the contents of a shot.
 - Preconditions: The hand is holding the shot; the shot contains the beverage.
 - Effects: The shot becomes empty.
- 6. clean-shot (shot, beverage, hand1, hand2): Cleans a used, empty shot.
 - **Preconditions**: hand1 holds the shot; hand2 is empty; the shot is empty and was previously used with the beverage.
 - Effects: The shot becomes clean and is no longer considered used.
- 7. pour-shot-to-clean-shaker(shot, ingredient, shaker, hand, level-prev, level-next): Pours an ingredient from a shot into a clean, empty shaker.
 - **Preconditions**: The hand holds the shot containing the ingredient; the shaker is empty and clean; the shaker is at level-prev.
 - Effects: The shot becomes empty; the shaker now contains the ingredient, is no longer empty or clean, becomes unshaken, and its fill level increases to level-next.
- 8. pour-shot-to-used-shaker(shot, ingredient, shaker, hand, level-prev, level-next): Adds a second ingredient to an unshaken shaker.
 - **Preconditions**: The hand holds the shot containing the ingredient; the shaker is unshaken and contains one ingredient; the shaker is at level-prev.
 - Effects: The shot becomes empty; the shaker now contains the additional ingredient; the shaker's fill level increases to level-next.
- 9. empty-shaker(hand, shaker, cocktail, level-prev, level-next): Empties a shaken cocktail from the shaker.
 - Preconditions: The hand holds the shaker; the shaker contains a shaken cocktail; the shaker is at level-prev.
 - Effects: The shaker becomes empty and unshaken; its fill level resets to level-next (empty).
- 10. clean-shaker (hand1, hand2, shaker): Cleans an empty shaker.
 - Preconditions: hand1 holds the shaker; hand2 is empty; the shaker is empty.
 - Effects: The shaker becomes clean.
- 11. **shake(cocktail, ing1, ing2, shaker, hand1, hand2)**: Mixes two ingredients in a shaker to create a cocktail.
 - **Preconditions**: hand1 holds the shaker; hand2 is empty; the shaker contains exactly ing1 and ing2; the shaker is unshaken.
 - Effects: The shaker becomes shaken; it now contains the resulting cocktail instead of the separate ingredients.
- 12. pour-shaker-to-shot (cocktail, shot, hand, shaker, level-prev, level-next): Serves a shaken cocktail from a shaker into a shot.
 - **Preconditions**: The hand holds the shaker containing the shaken cocktail; the shot is empty and clean; the shaker is at level-prev.
 - Effects: The shot now contains the cocktail and is no longer empty or clean; the shaker's fill level decreases to level-next.

A.2 ELEVATOR DOMAIN

The Elevator domain involves operating a set of elevators (fast and slow) to transport passengers between floors in a building. Each elevator has a specific capacity and can only access a defined set of floors. The goal is to move all passengers from their origin floors to their destination floors.

ACTIONS

756

758

759

760

761

762

764

765

766

767

768

769

770

771

772

773

774

775

776

777

778

779

781

782

783 784

785 786

787

788 789

790

791

792 793

794

795 796

797

798

799

800

801

802

803

804

805

806

807

808

809

- 1. move-up-slow (elevator, floor-from, floor-to): Moves a slow elevator up.
 - **Preconditions**: The elevator is at floor-from; floor-to is above floor-from; the elevator can reach floor-to.
 - Effects: The elevator is now at floor-to.
- 2. move-down-slow(elevator, floor-from, floor-to): Moves a slow elevator down
 - **Preconditions**: The elevator is at floor-from; floor-to is below floor-from; the elevator can reach floor-to.
 - **Effects**: The elevator is now at floor-to.
- move-up-fast (elevator, floor-from, floor-to): Moves a fast elevator up.
 - **Preconditions**: The elevator is at floor-from; floor-to is above floor-from; the elevator can reach floor-to.
 - **Effects**: The elevator is now at floor-to.
- 4. move-down-fast (elevator, floor-from, floor-to): Moves a fast elevator down.
 - **Preconditions**: The elevator is at floor-from; floor-to is below floor-from; the elevator can reach floor-to.
 - **Effects**: The elevator is now at floor-to.
- board (passenger, elevator, floor, count-prev, count-next): A passenger boards an elevator.
 - **Preconditions**: The passenger and elevator are at the same floor; the elevator's passenger count is count-prev; the elevator has capacity for another passenger.
 - Effects: The passenger is now on board the elevator; the elevator's passenger count becomes count-next.
- 6. leave(passenger, elevator, floor, count-prev, count-next): A passenger leaves an elevator.
 - **Preconditions**: The passenger is on board the elevator; the elevator is at the specified floor; the elevator's passenger count is count-prev.
 - Effects: The passenger is now at the floor; the elevator's passenger count becomes count-next.

A.3 PARKING DOMAIN

The Parking domain involves rearranging cars parked at curbs. Each curb can hold at most two cars: one parked at the curb and one double-parked behind it. A car cannot move if another car is parked behind it.

KEY PREDICATES

- clear (car): True if no car is double-parked behind this car.
- clear (curb): True if the curb is empty.

ACTIONS

- 1. move-curb-to-curb (car, curb-from, curb-to): A single-parked car moves to an empty curb.
 - Preconditions: car is at curb-from; car is clear; curb-to is clear.
 - Effects: curb-from becomes clear; car is now at curb-to, which is no longer clear.
- 2. move-curb-to-car(car-move, curb-from, car-ahead): A single-parked car double-parks behind another car.
 - Preconditions: car-move is at curb-from; car-move is clear; car-ahead is clear.
 - Effects: curb-from becomes clear; car-move is now behind car-ahead; car-ahead is no longer clear.
- 3. move-car-to-curb (car-move, car-ahead, curb-to): A double-parked car moves to an empty curb.
 - Preconditions: car-move is behind car-ahead; car-move is clear; curb-to is clear.
 - Effects: car-ahead becomes clear; car-move is now at curb-to, which is no longer clear.
- 4. move-car-to-car(car-move, car-from, car-to): A double-parked car moves to double-park behind a different car.
 - Preconditions: car-move is behind car-from; car-move is clear; car-to is clear.
 - Effects: car-from becomes clear; car-move is now behind car-to; car-to is no longer clear.

A.4 TETRIS DOMAIN

The Tetris domain involves moving and rotating Tetris pieces on a grid. Pieces can be one-square, two-square straight, or three-square L-shaped. A piece can only move or rotate into adjacent positions that are clear (empty).

ACTIONS

810

811 812

813

814 815

816 817

818

819

820

821 822

823

824

825

826 827

828

829

830

831

832

833

834

835

836

837

838

839

840 841

842 843

844

845

846 847

848

849

850

851

852

853

854

855

856

857

858

859

860

861

862

863

- 1. move_square (pos-from, pos-to, piece): Moves a one-square piece.
 - **Preconditions**: piece occupies pos-from; pos-to is clear; pos-from and pos-to are adjacent.
 - Effects: pos-from becomes clear; pos-to is now occupied by piece.
- move_two (pos-old, pos-pivot, pos-new, piece): Moves or rotates a two-square piece.
 - **Preconditions**: piece occupies pos-old and pos-pivot; pos-new is clear; pos-pivot and pos-new are adjacent.
 - Effects: pos-old becomes clear; pos-new is now occupied by piece.
- move_l_right (pA, pB, pC, pD, pE, pMid, piece): Moves or rotates an L-piece right.
 - **Preconditions**: piece occupies pA, pB, pC; pD, pE are clear; positions are correctly adjacent for a rightward move.
 - Effects: pA, pB become clear; pD, pE are now occupied by piece.
- 4. move_l_left (pA, pB, pC, pD, pE, piece): Moves or rotates an L-piece left.
 - **Preconditions**: piece occupies pA, pB, pC; pD, pE are clear; positions are correctly adjacent for a leftward move.
 - Effects: pA, pC become clear; pD, pE are now occupied by piece.
- 5. move_l_up (pA, pB, pC, pD, pE, pMid, piece): Moves or rotates an L-piece up.
 - **Preconditions**: piece occupies pA, pB, pC; pD, pE are clear; positions are correctly adjacent for an upward move.
 - Effects: pB, pC become clear; pD, pE are now occupied by piece.
- 6. move_1_down (pA, pB, pC, pD, pE, piece): Moves or rotates an L-piece down.
 - **Preconditions**: piece occupies pA, pB, pC; pD, pE are clear; positions are correctly adjacent for a downward move.
 - Effects: pA, pC become clear; pD, pE are now occupied by piece.

A.5 FLOOR TILES DOMAIN

The Floor Tiles domain involves robots painting a grid of tiles. Each robot can hold one color at a time and moves between adjacent tiles. A robot can paint an adjacent tile (above or below) with its current color, provided the tile is clear. Once a tile is painted, it cannot be occupied.

ACTIONS

- 1. change-color (robot, color-from, color-to): A robot changes its held paint color.
 - Preconditions: The robot is holding color-from; color-to is an available color.
 - **Effects**: The robot is now holding color-to.
- 2. paint-up(robot, tile-paint, tile-robot, color): A robot paints the tile above its current position.
 - **Preconditions**: robot is at tile-robot; tile-paint is directly above tile-robot; tile-paint is clear; robot is holding color.
 - Effects: tile-paint is now painted with color and is no longer clear.
- 3. paint-down(robot, tile-paint, tile-robot, color): A robot paints the tile below its current position.
 - **Preconditions**: robot is at tile-robot; tile-paint is directly below tile-robot; tile-paint is clear; robot is holding color.
 - Effects: tile-paint is now painted with color and is no longer clear.
- 4. up (robot, tile-from, tile-to): A robot moves one tile up.
 - **Preconditions**: robot is at tile-from; tile-to is directly above tile-from; tile-to is clear.
 - Effects: robot is now at tile-to; tile-from becomes clear; tile-to is no longer clear.
- 5. down(robot, tile-from, tile-to): A robot moves one tile down.

- **Preconditions**: robot is at tile-from; tile-to is directly below tile-from; tile-to is clear.
- Effects: robot is now at tile-to; tile-from becomes clear; tile-to is no longer clear.
- 6. right (robot, tile-from, tile-to): A robot moves one tile right.
 - **Preconditions**: robot is at tile-from; tile-to is directly to the right of tile-from; tile-to is clear.
 - Effects: robot is now at tile-to; tile-from becomes clear; tile-to is no longer clear.
- 7. left(robot, tile-from, tile-to): A robot moves one tile left.
 - **Preconditions**: robot is at tile-from; tile-to is directly to the left of tile-from; tile-to is clear.
 - Effects: robot is now at tile-to; tile-from becomes clear; tile-to is no longer clear.

B OVERVIEW OF PROMPTS

In this section, we provide a mapping between the main inference functions and their associated prompts, clarifying how the model is prompted at each stage of the search and diagrammatic reasoning pipeline. For each function, we specify: (1) the function name, (2) its core functionality, (3) where it is called in the search process, (4) the key points of its prompt, and (5) its input parameters. This overview enables readers to understand how the model is guided through each step of multimodal reasoning.

Function Name	Functionality	Where Called	Key Prompt Points	Input Parameters (Description)
next _action	Selects next best action, generates rea- soning, action, and new state description.	During child state generation (gener- ate_child_states).	- Presents problem, initial, current, and goal state (text and images) - Lists possible actions - Requests reasoning, action, and new state in code blocks - Handles uniqueness and previous errors if present	- Problem description (text) - Possible actions (list) - Initial, current, goal state objects (text, diagrams, schemas) - Model name, temperature - Chosen actions - Previous attempt/error (optional)
generate _diagram _schema	Generates diagram schema for a child state after an action.	During child state diagram generation (gen- erate_diagrams).	- Presents problem, initial, current, and new state (text, diagrams, schemas) - Requests one statement per object (position, size, status, identifier) - Enforces object count consistency - Handles previous attempt/error if present	- Problem description (text) - Initial state object (with schema) - Child state object (with action, parent, state description) - Model name, temperature - Previous at- tempt/error (optional)
test _diagram _schema	Verifies correctness of a generated diagram schema for a child state.	After schema generation, before code generation (generate_diagrams).	- Presents problem, initial, current, and child state (text, diagrams, schemas) - Validates object count, affected object status, and consistency - Requires yes/no answer and error summary	- Problem description (text) - Initial state object (with schema) - Child state object (with schema, action, parent) - Model name

Function Name	Functionality	Where Called	Key Prompt Points	Input Parameters (Description)
generate _diagram _code	Generates Matplotlib code to vi- sualize a child state based on its schema.	After schema validation (generate_diagrams).	- Presents problem, initial/current/child state (text, diagrams, schemas) - Provides example code and reasoning if available - Requests code to visualize all objects as described in schema - Enforces clarity, no overlaps, correct labeling, plausibility - Handles previous attempt/error if present	- Domain name - Problem description (text) - Child state object (with schema, action, parent) - Initial state object (with code, diagram) - Model name, temperature - Save path for image - Previous at- tempt/error (optional)
test _diagram	Verifies correctness and clarity of a generated diagram image for a child state.	After code execution, before accepting diagram (generate_diagrams).	- Presents problem, current/child state (text, diagrams, schemas) - Validates object presence, status, labeling, readability, plausibility - Requires yes/no answer and error summary	- Problem description (text) - Child state object (with schema, diagram, parent) - Domain name - Model name
check _action _validity	Checks if a proposed action and resulting state are valid (local verification).	After diagram generation, before path/global verification (generate_child_states).	- Presents problem, initial/current/goal state (text, diagrams, schemas) - Lists action path, action taken, new state description, and diagram - Checks preconditions, effects, diagram accuracy, reasoning - Requires yes/no answer and error summary	- Problem description (text) - Current state object (with schema, diagram) - Action taken (string) - New state description (text) - New state object (with schema, diagram) - Goal state object - Possible actions (list) - Initial state object - Model name
check _action _path	Checks if the entire action path from initial to current state is valid (global verification).	After local verification, before accepting child state (generate_child_states).	- Presents problem, initial/current/goal state (text, diagrams, schemas) - Lists full action path and parent state - Checks preconditions, effects, diagram accuracy for the whole path - Requires yes/no answer and reasoning	- Problem description (text) - Initial state object (with schema, diagram) - Current state object (with schema, diagram, parent) - Goal state object - Actions (list) - Possible actions (list) - Model name

Function Name	Functionality	Where Called	Key Prompt Points	Input Parameters (Description)
rank	Ranks can-	After child	- Presents problem,	- List of state objects
_states	didate states at	state generation,	goal state, and all can-	(with id, description,
	a given depth	before pruning	didate states (text, di-	diagram, action path)
	by proximity	(beam_search).	agrams, action paths)	- Problem
	to the goal state (for		- Requests ranking based	description (text) - Goal state object
	beam search).		on number of goal	- Model name
			constraints satisfied	
			- Requires	
			ranking code	
	C	A 4	block and reasoning	D 1.1
ini_g _diagrams	Generates and verifies	At the start of search,	- Presents problem, initial/goal state (text)	- Problem name, domain name
_dragrams	diagrams	during problem	- Requests diagram	- Problem
	for initial and	setup (setup).	schema and code	description (text)
	goal states.	1 \ 17	for initial/goal state	- Initial state
			- Enforces object	object (with text)
			coverage, status, and	- Goal state
			visual consistency - Verifies schema and	object (with text) - Model
			diagram correctness	name, temperature
			- Handles retries	- Parameters
			and error feedback	(e.g., max attempts)
				- Error message, previ-
	G 1'	XX7'.1 '	D . 11	ous attempt (optional)
generatediagram	Generates diagram schema	Within ini_g_diagrams	 Presents problem, initial and goal 	- Problem description (text)
_schema	for initial	(problem setup).	state descriptions	- Initial state
_ini_g	or goal state.	(F	- Requests	object (with text)
			one statement	- Goal state
			per object (position,	object (with text)
			size, status, identifier) - Handles previous at-	- Model
			tempt/error if present	name, temperature - Domain name
			tempa error ir present	- Goal flag (bool)
				- Previous at-
				tempt/error (optional)
test	Verifies	Within	- Presents problem,	- Problem
_diagram _schema	correctness	ini_g_diagrams	state description,	description (text)
_ini_g	of initial/goal state diagram	(after schema generation).	and schema - Validates object	- Initial state object (with schema)
-	schema.	Scheration).	coverage and status	- Goal state
			- Requires	object (with schema)
			yes/no answer	- Model name
			and error summary	- Goal flag (bool)
generate	Generates	Within	and error summary - Presents problem,	- Goal flag (bool) - Domain name
_diagram	Matplotlib	ini_g_diagrams	and error summary - Presents problem, state description,	- Goal flag (bool) - Domain name - Problem
_	Matplotlib code for initial	ini_g_diagrams (after schema	- Presents problem, state description, and schema	- Goal flag (bool) - Domain name - Problem description (text)
_diagram _code	Matplotlib code for initial or goal state	ini_g_diagrams	and error summary - Presents problem, state description, and schema - Provides	- Goal flag (bool) - Domain name - Problem description (text) - Initial state
_diagram _code	Matplotlib code for initial	ini_g_diagrams (after schema	and error summary - Presents problem, state description, and schema - Provides code and reasoning - Requests code to	- Goal flag (bool) - Domain name - Problem description (text) - Initial state object (with schema) - Goal state
_diagram _code	Matplotlib code for initial or goal state	ini_g_diagrams (after schema	and error summary - Presents problem, state description, and schema - Provides code and reasoning - Requests code to visualize all objects as	- Goal flag (bool) - Domain name - Problem description (text) - Initial state object (with schema) - Goal state object (with schema)
_diagram _code	Matplotlib code for initial or goal state	ini_g_diagrams (after schema	and error summary - Presents problem, state description, and schema - Provides code and reasoning - Requests code to visualize all objects as described in schema	- Goal flag (bool) - Domain name - Problem description (text) - Initial state object (with schema) - Goal state object (with schema) - Model
_diagram _code	Matplotlib code for initial or goal state	ini_g_diagrams (after schema	and error summary - Presents problem, state description, and schema - Provides code and reasoning - Requests code to visualize all objects as described in schema - Enforces clarity,	- Goal flag (bool) - Domain name - Problem description (text) - Initial state object (with schema) - Goal state object (with schema) - Model name, temperature
_diagram _code	Matplotlib code for initial or goal state	ini_g_diagrams (after schema	and error summary - Presents problem, state description, and schema - Provides code and reasoning - Requests code to visualize all objects as described in schema - Enforces clarity, labeling, plausibility	- Goal flag (bool) - Domain name - Problem description (text) - Initial state object (with schema) - Goal state object (with schema) - Model name, temperature - Save path for image
_diagram _code	Matplotlib code for initial or goal state	ini_g_diagrams (after schema	and error summary - Presents problem, state description, and schema - Provides code and reasoning - Requests code to visualize all objects as described in schema - Enforces clarity, labeling, plausibility - Handles previous	- Goal flag (bool) - Domain name - Problem description (text) - Initial state object (with schema) - Goal state object (with schema) - Model name, temperature - Save path for image - Goal flag (bool)
_diagram _code	Matplotlib code for initial or goal state	ini_g_diagrams (after schema	and error summary - Presents problem, state description, and schema - Provides code and reasoning - Requests code to visualize all objects as described in schema - Enforces clarity, labeling, plausibility	- Goal flag (bool) - Domain name - Problem description (text) - Initial state object (with schema) - Goal state object (with schema) - Model name, temperature - Save path for image

Function Name	Functionality	Where Called	Key Prompt Points	Input Parameters (Description)
test	Verifies	Within	- Presents problem,	- Problem
_diagram	correctness	ini_g_diagrams	state description,	description (text)
_ini_g	and clarity	(after	and diagram image	- Initial
	of initial/goal	code execution).	- Validates object	state object (with
	state diagram		presence, status,	schema, diagram)
	image.		labeling, consistency,	- Goal
			plausibility	state object (with
			- Requires yes/no answer	schema, diagram) - Domain name
			and error summary	- Model name
			and crioi summary	- Goal flag (bool)
is_unique	Checks	During child	- Presents problem,	- Problem
action	if a proposed	state generation	current state,	description (text)
	action leads to	in gener-	action taken, and	- Current state object
	a unique child	ate_child_states.	new state description	(with child states)
	state from the		- Lists previously	- Action taken (string)
	current state.		explored actions	- New state
			and resulting states	description (text)
			- Asks if the new	- Model name
			action/state is unique	
			- Requires yes/no answer and explanation	
-11	Checks	At each	- Presents	- Problem
check _goal	if the current	node expansion	problem, initial,	description (text)
_goar _state	state satisfies	in beam search.	current, and goal	- Current state
_	all goal state	in count_courcil.	state (text, diagrams)	object (with diagram)
	constraints.		- Lists action path	- Goal state
			- Asks if	object (with diagram)
			current state matches	- Initial state
			all goal constraints	object (with diagram)
			- Requires yes/no	- Model name
			answer and step-	
			by-step reasoning	

This mapping clarifies the modular structure of the inference pipeline and the precise role of each prompt in guiding the model's multimodal reasoning. Each function is responsible for a distinct aspect of the search or verification process, and the prompts are carefully designed to enforce correctness, clarity, and consistency at every stage. The table above can be used as a reference for understanding or extending the codebase, as well as for reproducing or adapting the prompting strategy to new domains or tasks.

C CODE STRUCTURE AND ORGANIZATION

The codebase is organized to support multimodal planning and diagrammatic reasoning across multiple domains. Each domain (e.g., blocksworld, barman, elevator, parking, tetris, tiles) is self-contained, with a consistent folder structure and supporting scripts. Below, we describe the main components and their roles.

C.1 TOP-LEVEL STRUCTURE

- Domain Folders: Each domain (e.g., blocksworld, barman, etc.) is a top-level folder containing all files and subfolders needed to run our method on the instances of that domain.
- Shared Scripts: At the root, scripts such as search.py, inference.py, and diagram generation scripts are provided for general use across domains.
- Utilities: Files like requirements.txt and Readme.md provide environment setup and documentation.

C.2 KEY SCRIPTS AND THEIR ROLES

- **search.py**: The main search and script. Implements the graph of thought algorithm with beam search and backtracking, manages state expansion, diagram generation, and verification. For each problem instance, it creates a subfolder (e.g., blocksworld_instance_1/) and stores all intermediate and final results, including state diagrams, code, and logs.
- **inference.py**: Contains all functionalities facilitated by LMMs and prompting logic, including functions for generating and verifying actions, diagram schemas, code, and state rankings. This script is the interface between the search process and the language model.
- visual_thinking.py: Main pipeline script. Orchestrates the batch processing of multiple problem instances for a given domain. For each instance, it sets up the directory structure, runs our diagrammatic search, and validates the resulting plan using VAL. It also manages the output and validation logs.
- initial_diagram_schema.py, initial_diagram_code.py, initial_goal_diagram_code.py: Scripts for generating and storing the first initial and goal conceptual diagrams for the domain and their schemas. Outputs are stored in the initial_conceptual_diagram/ folder.
- PDDL_tranlation.py (per domain): Handles translation between natural language and PDDL for that domain, including prompt templates and in context examples.

C.3 INSTANCE AND STATE FOLDER STRUCTURE

For each problem instance (e.g., blocksworld_instance_1/), the following structure is created during search:

- state X/: For each explored state, a folder is created containing:
 - diagram.png: The generated diagram for the state.
 - diagram_code.py: The code used to generate the diagram.
 - diagram_schema.txt: The schema describing the diagram.
 - info.txt: Metadata about the state, including parent, action taken, and reasoning.
 - attempts/: Subfolders for storing all attempts at generating child states and diagrams, including error logs.
- goal_state/: If a goal is found, this folder contains the final state information and a copy of all diagrams along the solution path.
- ranking/: Stores state ranking information at each search depth.
- output.txt, plan.pddl, val_output.txt: Output logs, the generated plan, and validation results.

C.4 DOMAIN EXAMPLES

The repository attached in the supplementary material includes several PDDL domains (blocksworld, barman, elevator, parking, tetris, tiles), each with a complete set of PDDL files, instance generators, randomly generated instances, and translations of the domain and a random state and initial conceptual diagrams generated using pipeline. For blocksworld, we also provide full examples of solved problems, including all intermediate diagrams and logs, to facilitate reproducibility and intuitive understanding of the pipeline. To view the sequence of diagrams produced for a successfully solved instance, refer to the goal_state folder within each instance's directory. This folder contains the chain of diagrams generated by the pipeline from the initial state to the goal state for that instance.

This modular structure allows for easy extension to new domains and facilitates reproducibility, inspection, and further research.