

# Prime the Search: Using Large Language Models for Guiding Geometric Task and Motion Planning by Warm-starting Tree Search

Journal Title  
XX(X):1–11  
©The Author(s) 2024  
Reprints and permission:  
sagepub.co.uk/journalsPermissions.nav  
DOI: 10.1177/ToBeAssigned  
www.sagepub.com/

SAGE

Dongryung Lee<sup>\*1</sup>, Sejune Joo<sup>\*1</sup>, Kimin Lee<sup>1</sup>, and Beomjoon Kim<sup>1</sup>

## Abstract

The problem of relocating a set of objects to designated areas amidst movable obstacles can be framed as a Geometric Task and Motion Planning (G-TAMP), a subclass of task and motion planning problem (TAMP) [1]. Traditional approaches to G-TAMP have relied either on domain-independent heuristics [2] or on learning from planning experience [1, 3–7] to guide the search, both of which typically demand significant computational resources or data. In contrast, humans often use common sense to intuitively decide which objects to manipulate in G-TAMP problems. Inspired by this, we propose leveraging Large Language Models (LLMs), which have common sense knowledge acquired from internet-scale data, to guide task planning in G-TAMP problems. To enable LLMs to perform geometric reasoning, we design a predicate-based prompt that encodes geometric information derived from a motion planning algorithm. We then query the LLM to generate a task plan, which is then used to search for a feasible set of continuous parameters. Since LLM is prone to mistakes [8], instead of committing to LLM's outputs we extend Monte Carlo Tree Search (MCTS) to a hybrid action space and use the LLM to guide the search. Unlike the previous approach [9] that calls an LLM at every node and incurs high computational costs, we use it to warm-start the MCTS with the nodes explored in completing the LLM's task plan. On six different G-TAMP problems, we show our method outperforms previous LLM planners and pure search algorithms.

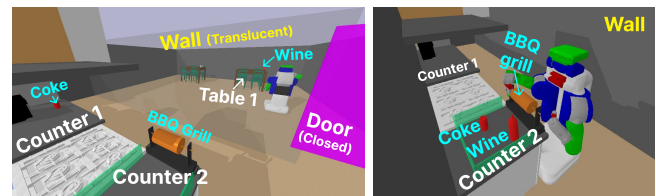
## Keywords

Task and Motion Planning, Large Language Models, Monte Carlo Tree Search

## 1 Introduction

Imagine a robot operating in a restaurant tasked with bringing items to a kitchen. Ideally, you would directly fetch target objects, but there often are obstacles in the way as shown in Figure 1. To solve the problem, the robot must figure out a sequence of objects and motions to clear obstacles and bring the goal objects to desired regions. These problems can be formulated as a geometric task and motion planning (G-TAMP) problem, a complex problem that involves hybrid action space that includes discrete actions, such as selecting which skill to use and which object to manipulate, as well as continuous actions, like determining the specific manipulation motion for each object. Additionally, the problem involves intricate reachability constraints among the movable obstacles.

Currently, there are two main approaches to G-TAMP problems. The first is pure-planning algorithms [2], which typically integrate classical AI planning algorithms [10] that use a domain-independent heuristic function with sampling or optimization to handle continuous parameters. While effective for general TAMP problems, this approach struggles with utilizing domain-specific information to identify the cause of infeasibility. For example, to determine whether the door needs to be opened in Figure 1, it must first attempt to plan a motion across the door, and only upon failure seek alternative discrete actions, that *may* open the door. This process is highly inefficient, often requiring numerous



**Figure 1.** An example of a G-TAMP problem. **Left:** The initial configuration. The robot must bring the wine from Table 1, Coke from Counter 1, and a BBQ grill to Counter 2. The grill is already in its goal position but obstructing the placement of other goal objects, and a closed door between the kitchen and dining area must be opened to traverse these two areas. **Right:** A goal configuration. To achieve this, the robot must open the door, temporarily remove the grill from Counter 2, place other objects, and then bring back the grill.

motion planning calls to identify the source of the failure. It would be much more efficient to perform causal reasoning such as “because the door is closed, and the door is in the

<sup>1</sup>Graduate School of AI, Korea Institute of Advanced Science and Technology

### Corresponding author:

Beomjoon Kim, Korea Institute of Advanced Science and Technology, Graduate School of AI, Seoul, Korea.

Email: beomjoon.kim@kaist.ac.kr

<sup>\*</sup>co-first authors

way of moving the wine to Counter 2, we need to open the door.”

Alternatively, we can use learning to guide search from planning experience. In particular, several works learn domain-specific heuristic functions or constraints to guide *task planning*, which is planning a sequence of discrete actions [4, 5, 11, 12]. They have been shown to considerably improve planning speed compared to pure planning strategies because they can learn through correlation. For example, in all successful plans, the door was opened in states where the robot had to move objects to the kitchen from the dining area, so we must open the door now. The problem, however, is that they typically require a significant amount of planning experience to acquire such knowledge, which is time-consuming to collect.

Our observation is that the knowledge required for task planning in a G-TAMP problem is straightforward for an agent with common sense, provided that the problem is expressed with the abstract representation that clearly encodes the constraints and goals. For example, in Figure 1, humans intuitively understand that the door needs to be opened if they know that the door is in the way of a manipulation motion. Based on this observation, we propose to use LLMs pre-trained on internet data for task planning, as they likely possess such common sense without additional training. The main challenge, however, lies in designing prompts that are effective across diverse problems and in managing erroneous outputs from an LLM caused by incorrect reasoning or hallucination [8, 13].

One approach for designing the prompt is to adopt that of SayCan [14], which consists of a task instruction, action history, and example plan. However, because it lacks state information, it is difficult to perform state-based causal reasoning. There also are several prompt designs that include predicate-based state information [15–17]. However, they typically lack geometric information such as whether an obstacle is in the path to a particular object, which can only be evaluated via motion planning algorithms.

To solve this, we propose to use *geometric predicates*, which have shown to be effective in learning a relational value function for G-TAMP problems [1, 11], in our prompt to represent goals, states, and domains. As in previous work, we use a motion planning algorithm to compute these which encode reachability and occlusion. While our prompt can take various formats, we use the PDDL-style format for its clarity and proven efficacy with LLMs [15, 18].

To combat the LLM’s prediction errors [13, 16], we propose integrating an LLM with tree search, so that we can explore actions beyond those suggested by the LLM. The critical design choice here is how to structure the interface between the tree search and LLM. One approach is to invoke the LLM at every node. For example, LLM-MCTS [9] combines an LLM with MCTS by deriving a policy from a batch of LLM responses and using it to determine which action to explore first. However, this method is highly inefficient since it calls the LLM at every node, and each LLM call involves processing a long sequence of tokens containing objects, states, and domain descriptions, with computation scaling quadratically to token length. Furthermore, MCTS is only restricted to discrete action spaces and is not applicable to G-TAMP.

To solve this, we first extend MCTS to hybrid action spaces and propose a method called **Search Tree augmented by Language Model (STaLM)**, which uses an LLM to warm-start an MCTS. STaLM first queries the LLM for a batch of task plans that give discrete action choices but not continuous parameters. It concretizes these plans by searching for a feasible sequence of continuous parameters, and if this fails, initiates an MCTS that has been warm-started with the nodes that have been explored while trying to concretize the LLM’s plans. Our intuition is that by concretizing batch queried task plans and using them for warm-started MCTS, the number of LLM queries can be minimized while leveraging the common sense from LLM to avoid numerous motion planning calls to identify the source of the failure. Figure 2 demonstrates our method.

In six different G-TAMP problems, we demonstrate that our prompt design is more effective than the existing prompt designs, and show that STaLM is more computationally efficient than the state-of-the-art pure planning algorithms or other LLM-based planning algorithms.

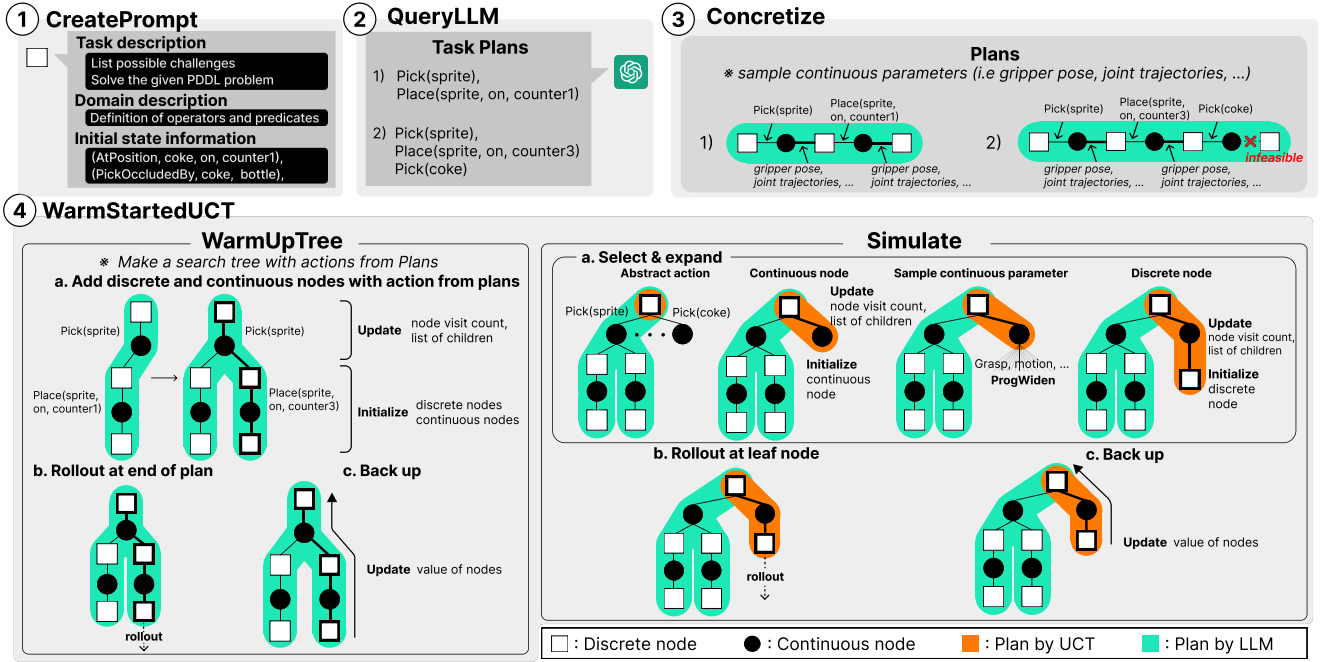
## 2 Related work

### 2.1 Task and motion planning

Task and motion planning (TAMP) is a class of planning problems that require integrated discrete task planning and continuous motion planning [2]. One approach to TAMP is computing a task plan and refining it via sampling or optimization. Here, a task plan is a sequence of symbolic actions, which refer to an action with its continuous parameters unspecified. For example, PDDLStream [19] creates a task plan with a classical planning system [10] and refines the plan by sampling the continuous parameters with external functions called streams. eTAMP [20] uses tree search [21] to sample continuous parameters for the task plans from top-k planner [22]. Logic Geometric Programming (LGP) [23] frames TAMP as a continuous mathematical program and explicitly aims to optimize the final configuration represented in an objective function. LGP solvers use top-k-planner [24], tree search with hierarchical relaxation of constraints [25], or tree search guided by hand-designed heuristics [26] to compute a task plan. They then solve continuous optimization problems with each symbolic action in the given task plan as a constraint to obtain continuous parameters. However, these methods lack common sense and require performing a search to compute a plan even for trivial matters like “pick the occluding object”, rendering them inefficient.

### 2.2 Learning to guide task and motion planning

There have been several attempts to adopt the intuitions from AlphaGo [27–29] to G-TAMP problems. Several studies propose to guide task planning by training a heuristic function based on images [5, 30], a graph that encodes the grounded predicates expressing the relationship among objects [1, 4, 31], or context-dependent abstractions [32]. While a raw image does not explicitly include reachability and occlusion, Kim et al. [1] directly use predicates about occlusions. Inspired by this, we also encode the state



**Figure 2.** Overview of STaLM. (1) **CreatePrompt** uses a motion planning algorithm to compute state information for the prompt, such as one in Figure 3, for querying the LLM. (2) **QueryLLM** uses the prompt to generate  $N_{batch}$  number of task plans, which are a sequence of discrete actions such as `Pick(sprite)`, `Place(sprite, on, counter1)`. (3) **Concretize** concretizes the given task plans by searching a feasible sequence of continuous parameters (e.g. gripper pose, joint trajectories, etc) for each discrete action. The squares denote discrete nodes for discrete action choices and circles denote continuous nodes for continuous parameter choices. After we have both discrete and continuous decisions, we simulate the next state by applying it and moving on to the next discrete node. If we succeed in finding a feasible concrete plan, we return the plan, otherwise, we commence the next step. (4) **WarmStartedUCT** consists of two processes: **WarmUpTree** and **Simulate**. **WarmUpTree** initializes the search tree with the states and actions explored in step (3), where the values of leaf nodes are estimated via roll-out and backed up along the tree. **Simulate** uses the usual MCTS operations on the warm-started tree: node selection using UCT, expansion, rollout, and backup. The orange bubble shows nodes explored by tree search, and the green bubble shows nodes explored using LLM’s suggestions. We use Progressive Widening to expand at continuous nodes. See Appendix B for detailed pseudocode.

with a set of grounded geometric predicates. Other works train the samplers for continuous choices such as grasp or placement of the object using meta-learning [33] or generative models [6, 7, 34]. However, to train any form of function, a significant computational cost is required to collect training data. In contrast, we use a pre-trained LLM to guide the search without additional learning. There are several studies [35–39] that propose to learn predicates and operators, enabling more task-specific planning. Our work can be used in conjunction with these methods.

### 2.3 Planning with LLMs

Several methods [9, 14–16, 40–44] use LLMs for planning with distinct skills, constraints, and objectives. Given a task instruction, SayCan [14] outputs the probability of the next skill as a product of the LLM’s probability of using that skill and skill affordance, the probability of that skill succeeding in the current state. Yet, its prompt lacks state information necessary for state-based causal reasoning. To enable this, a line of work [15–17, 43, 44] provides predicate-based state information to the LLM. However, even with state information, several works have shown that LLMs are not perfect at planning and tend to produce erroneous outputs [8, 13]. To combat this, other works [13, 40] use LLM with self-refinement [45] to generate a new response by providing past output and feedback. However, this strategy also shows limited improvement on domains that require a significant amount of diversity and exploration [8, 40] since the choice

of actions is fundamentally limited to actions given by the LLM. LLM-MCTS [9] instead queries the LLM to obtain a policy to guide MCTS via PUCT [46] and explore beyond actions that are suggested by LLM. Yet, LLM-MCTS has to call the LLM for every simulation, which is extremely expensive. In contrast, our framework only queries the LLM at the initial state.

## 3 Method

### 3.1 Problem formulation

We consider an environment that consists of a set of movable objects  $\mathbf{O} = \{o_i\}_{i=1}^{n_O}$ , a set of regions  $\mathbf{R} = \{r_i\}_{i=1}^{n_R}$ , and a set of doors  $\mathbf{D} = \{d_i\}_{i=1}^{n_D}$ . We model the world with a deterministic Markov Decision Process (MDP) with a state space  $S$ , a parameterized action space  $\mathcal{A}$ , a deterministic transition model  $T$ , and reward function  $R$ . A state is defined by the stable poses of movable objects,  $\mathcal{P}_{o_i} \in SE(2)$ , whether the doors are opened or closed  $\mathcal{U}_{d_i} \in \{0, 1\}$ , and robot configuration  $c \in \mathcal{C}$ , and is denoted as  $s \in S$  where  $s = (\mathcal{P}_{o_1}, \dots, \mathcal{P}_{o_{n_O}}, \mathcal{U}_{d_1}, \dots, \mathcal{U}_{d_{n_D}}, c)$ . All entities have known and fixed shapes. An action  $\alpha(\delta, \kappa) \in \mathcal{A}$  consists of an operator name  $\alpha$ , such as `PLACE`, a set discrete parameters  $\delta$ , such as a region to place an object down, and a set of continuous parameters  $\kappa$ , such as a trajectory.

Each  $\alpha(\delta, \kappa)$  induces a mapping  $T(s, \alpha(\delta, \kappa))$  from  $s$  to  $s' \in S$ . If  $\alpha(\delta, \kappa)$  cannot be legally executed at  $s$ , we let  $s' = s$ , absorbing the “failure” state and terminating

the simulation. A goal  $\mathcal{G}$  is given as a conjunction of  $(\text{AtPosition } o_{\text{goal}}, \text{dir}, o_{\text{ref}})$ , where  $o_{\text{goal}} \in \mathbf{O}$ ,  $\text{dir} \in \{\text{on, left, right, front, behind}\}$ , and  $o_{\text{ref}} \in \mathbf{O} \cup \mathbf{R}$ . A problem instance consists of  $(\mathbf{O}, \mathbf{R}, \mathbf{D}, s_0, \mathcal{G}, T, R, h)$ , where  $s_0$  is the initial state and  $h$  is the planning horizon. The objective is to find a sequence of actions that satisfies  $\mathcal{G}$ .

We define an *action* as  $\mathbf{a}(\delta, \kappa)$ , which has a *discrete action*  $\mathbf{a}(\delta)$  and *continuous parameter*  $\kappa$ , and a sequence of  $\mathbf{a}(\delta)$  as a *task plan*. Our method consists of three main components: (1) a prompt design based on geometric predicates for querying the LLM for task plans, (2) a search algorithm that concretizes the task plan by finding a feasible set of continuous parameters, and (3) if unsuccessful, commences a hybrid action space Upper Confidence Tree (UCT) [47] warm-started with explored nodes.

### 3.2 Predicate computation and prompt design

Our prompt consists of a task instruction, domain description, goal, objects, and the initial state as shown in Figure 3. We use a set of geometric predicates such as `PICKOCCLUDED` and `PLACEOCCLUDED`, and use motion planning algorithms to compute them. Concretely, we first compute the path to pick or place the designated object without considering other movable objects. Then, we check collisions between the swept volume of the motions and movable objects. If there is a collision, we set the occlusion predicate as true.

Part of our **Task Instruction** (Fig 3 top) asks for possible challenges for the problem. Empirically, without this, we have observed that LLMs often fail to respect an action’s preconditions (e.g. pick the goal object despite being occluded by other objects). This was inspired by chain-of-thought (CoT) [48], which showed that LLM’s reasoning capability improves when asked to generate intermediate reasoning steps. Our insight is that since our problem is geometric, in which occlusion relations are of main concern, this will make the LLM to respect the preconditions. An example prompt of STaLM is included in Appendix F.

### 3.3 Search Tree augmented by Language Model (STaLM)

The key idea of STaLM is to prioritize the actions given by the LLM but complement the planning with additional search whenever in case LLM fails to provide the solution. Unlike the previous work which queries the LLM at every time step, we simply use the states we explored during the concretization of task plans to warm up the tree search. This reduces the number of expensive LLM calls, but still directs the search into a promising region of the search space.

Algorithm 1 gives a pseudocode for STaLM. It takes an initial state  $s_0$ , planning horizon  $h$ , LLM query batch size  $N_{\text{batch}}$ , and planning budget  $N_{\text{budget}}$  as inputs. The algorithm first queries the LLM for a batch of task plans with `CreatePrompt`, which creates our prompt from  $s_0$  and  $\mathcal{G}$  (L2), and concretizes the task plans via `Concretize`, which searches for a sequence of continuous parameters using sampling (L3). If any of the plans succeeds, we return the plan, otherwise, we commence `WarmStartedUCT`, an MCTS for hybrid action space. We pictorially explain how

Algorithm 1 works in Figure 2. More detailed pseudocode for each subroutine is included in Appendix B.

---

#### Algorithm 1: STaLM ( $s_0, h, N_{\text{batch}}, N_{\text{budget}}$ )

---

```

1 Global Variables:  $\mathcal{G}, H, \gamma$ 
2 TaskPlans  $\leftarrow$ 
   QueryLLM(CreatePrompt( $s_0, \mathcal{G}$ ),  $N_{\text{batch}}$ )
3 success, Plans  $\leftarrow$  Concretize (TaskPlans,  $s_0, h$ )
4 if success
5   return Plans
6  $Q \leftarrow$  WarmStartedUCT(Plans,  $s_0, h, N_{\text{budget}}$ ).Q
7 return  $\arg \max_{\mathbf{a}, \delta, \kappa} Q(s, \mathbf{a}(\delta, \kappa))$ 

```

---

## 4 Experiments

### 4.1 Experiment setup

We implement our domains that consist of two areas, a kitchen and a hall, separated by a door that must be opened to navigate the areas in PyBullet [49]. Each area has regions on which movable objects can be placed. We have the following assumptions: (1) the robot only uses its right arm for manipulation. (2) We have a pre-defined robot base pose  $(x_r, y_r, \psi_r) \in SE(2)$  for each region and door. (3) The robot grasps object  $o$  using a pre-defined grasp position  $(x_g^o, y_g^o, z_g^o)$ , and (4) the orientation of  $o$  in the robot frame,  $\omega_o$ , is fixed during placement.

We design six problems to test the capabilities of different algorithms as shown in Figure 4. There are  $n(\mathbf{R}) + 4 \times n(\mathbf{O})$  possible placements in our domains where 4 represents the four possible directions, left, right, front, and behind. For  $P1-6$ , the number of possible placement locations is 31, 23, 50, 27, 21, and 27 respectively. We perform 50 trials for  $P1$  to  $P6$  with max search depth  $h$  of 20 and a time budget of 300 seconds for  $P1, 2, 4$  and 600 seconds for  $P3, 5, 6$ .

The robot has 3 operators: `PICK`, `PLACE`, and `OPEN`. Table 1 summarizes each operator’s discrete and continuous parameters. The continuous parameters that consist of base trajectory  $\tau_{\text{nav}}$ , gripper position  $p_g$  and orientation  $\omega_g$ , and arm motion  $\tau_g$  for each operator are sampled by the following procedures.

- `PICK`

1. Use a motion planner, Probabilistic Roadmap (PRM), to compute  $\tau_{\text{nav}}$  from the current robot base pose to  $\mathcal{P}_b$ .
2. Gripper position  $p_g$  is given as the discrete parameter for `PICK`. With the base pose fixed at  $\mathcal{P}_b$ , uniformly sample the gripper orientation  $\omega_g$  from the pre-defined ranges of pitch  $\phi \in [0, 45^\circ]$  and yaw  $\chi \in [-60^\circ, 60^\circ]$  of the gripper.
3. Find an inverse kinematics (IK) solution for the gripper pose  $(p_g, \omega_g)$ . Compute  $\tau_g$  by linearly interpolating from the current robot arm joint configuration to the IK solution.

- `OPEN`

1. Use a motion planner (PRM) to compute  $\tau_{\text{nav}}$  from the current robot base pose to  $\mathcal{P}_b$ .




**Task Instruction**  
 You are an expert proficient in PDDL and planning actions for a problem. Your response should follow this template:  
**## Possible challenges for unachieved goals based on given state ##**  
**## Plan ##**  
 plan = [(['action\_type', 'args\_1', 'args\_2', ...],)]

**Domain Description (Predicates)**  
 (define (domain shop)  
 (:requirements :typing :derived-predicates)  
 (:types movable\_object region openable - object)  
 (:constants on left\_of right\_of front\_of behind\_of - direction)  
**(:predicates**  
**(RobotHolding ?o)** ; True if robot is holding movable\_object  
**(HandAvailable)** ; True if robot hand is available  
**(AtPosition ?s ?dir ?ref)** ; True if s is at direction of ref  
**(IsClosed ?o)** ; True if the door is closed  
**(PickOccludedBy ?o ?occ)** ; True if (pick, o) is occluded by occ.  
**(PlaceOccludedBy ?o ?dir ?ref ?occ)** ; True if (place, o, dir, ref) is occluded by occ  
  
 (:derived (UnsafePick ?s)  
 (exists (?occluder) ((PickOccludedBy ?s ?occluder))))  
 (:derived (UnsafePlace ?s ?dir ?ref)  
 (exists (?occluder) ((PlaceOccludedBy ?s ?dir ?ref ?occluder))))  
**(:end)**

**Domain Description (Actions)**  
**(:action pick**  
 :parameters (?o)  
 :precondition (and (HandAvailable) (not (UnsafePick ?o)))  
 :effect (and (not (HandAvailable)) (RobotHolding ?o))  
**(:action open**  
 :parameters (?o)  
 :precondition (and (IsClosed ?o) (HandAvailable))  
 :effect (and (not (IsClosed ?o)) (HandAvailable))  
**(:action place**  
 :parameters (?o ?dir ?ref)  
 :precondition (and (RobotHolding ?o)  
 (not (UnsafePlace ?o ?dir ?ref)))  
 :effect (and (not (RobotHolding ?o)) (HandAvailable)  
 (AtPosition ?o ?dir ?ref))  
**(:end)**

**(:objects**  
 salter - movable\_object,  
 bottle - movable\_object,  
 counter1 - region,  
 counter2 - region,  
 shelf - region,  
 table1 - region,  
 table2 - region,  
 kitchen\_door - door)  
**(:end)**



**Initial state information**  
 (:init  
 ((HandAvailable),  
 (AtPosition salter on counter2),  
 (AtPosition bottle1 on counter2),  
 (AtPosition salter behind\_of bottle1),  
 (AtPosition bottle1 front\_of salter),  
 (PickOccludedBy salter bottle),  
 (PlaceOccludedBy salter on counter1 bottle),  
 (PlaceOccludedBy salter on counter2 bottle),  
 (PlaceOccludedBy salter on shelf bottle),  
 (PlaceOccludedBy salter on table1 kitchen\_door),  
 (PlaceOccludedBy salter on table2 kitchen\_door),  
 (PlaceOccludedBy bottle on table1 kitchen\_door),  
 (PlaceOccludedBy bottle on table2 kitchen\_door),  
 (PlaceOccludedBy salter left\_of bottle bottle),  
 (PlaceOccludedBy salter right\_of bottle bottle),  
 (PlaceOccludedBy salter front\_of bottle bottle),  
 (PlaceOccludedBy salter behind\_of bottle bottle))  
**(:end)**

**Goal**  
 (:goal (AtPosition salter on sink))  
 Generate a plan to achieve the goals from init.

**Figure 3.** An example prompt for the state shown in the top-right corner (the image is not given to the LLM). **Task instruction** defines the output template and ask the LLM to state the challenges. **Domain Description (Predicates)** defines the set of predicates for our domain. **Domain Description (Actions)** gives operator definitions. The right column gives a set of entities, the initial state, and the goal defined by the predicates and the entities.

	Discrete parameters					Continuous parameters				
	$o_{target}$	$dir$	$o_{ref}$	$\mathcal{P}_b$	$p_g$	$\omega_g$	$\tau_{nav}$	$p_g$	$\omega_g$	$\tau_g$
PICK	$o_{target} \in \mathbf{O}$	.	.	$(x_r, y_r, \phi_r)$	$(x_{g_{target}}^{o_{target}}, y_{g_{target}}^{o_{target}}, z_{g_{target}}^{o_{target}})$	.	.	.	$(\phi, \chi)$	.
OPEN	$o_{target} \in \mathbf{D}$	.	.	$(x_r, y_r, \phi_r)$	$(x_{g_{target}}^{o_{target}}, y_{g_{target}}^{o_{target}}, z_{g_{target}}^{o_{target}})$	$\omega_{g_{target}}^{o_{target}}$	Motion planner	.	.	Motion planner
PLACE	$o_{target} \in \mathbf{O}$	on, left, right, front, behind	$o_{ref} \in \mathbf{O}$ if $dir \neq \text{on}$ $o_{ref} \in \mathbf{R}$ if $dir = \text{on}$	$(x_r, y_r, \phi_r)$	.	$\omega_{g_{target}}^{o_{target}}$	.	$(x_p, y_p)$	.	.

**Table 1.** Operator descriptions for PICK, OPEN, and PLACE. A dot indicates an unused parameter.  $o_{target}$  is the target object.  $dir$  and  $o_{ref}$  are the placement direction and reference object respectively. There is only one  $\mathcal{P}_b \in SE(2)$  for each target region  $r$ . For PICK and OPEN, there is a single gripper position  $p_g$ . In OPEN, we have the fixed gripper orientation  $\omega_g$  for the door. For PLACE, we compute  $\omega_g$  so that  $o_{target}$ 's orientation with respect to the region on which it is placed stays same as before the pick. Both base and arm motions,  $\tau_{nav}$  and  $\tau_g$  respectively, are computed using motion planners and apply to all operators. For PLACE,  $p_g = (x_p, y_p)$  is randomly sampled from the placement region's surface. For PICK,  $\omega_g$  consists of gripper's pitch  $\phi$  and yaw  $\chi$ , randomly sampled from specified ranges.

- Both  $p_g$  and  $\omega_g$  are given as the discrete parameters for OPEN. Find an IK solution for  $(p_g, \omega_g)$  and compute  $\tau_g$  by linear interpolation.

- PLACE

- Use a motion planner (PRM) to compute  $\tau_{nav}$  from the current robot base pose to  $\mathcal{P}_b$ .
- $\omega_g$  is given as a discrete parameter for PLACE. With the base pose fixed at  $\mathcal{P}_b$ , uniform-randomly sample  $p_g = (x_p, y_p)$  in the region for placement. We use rejection sampling until the direction of  $p_g$  aligns with  $dir$  with respect to  $o_{ref}$ .
- Find an IK solution for  $(p_g, \omega_g)$  and compute  $\tau_g$  by linear interpolation.







If we fail to sample feasible  $\tau_{nav}$ ,  $p_g$ ,  $\omega_g$ , and  $\tau_g$  within a fixed number of trials, we mark the action

as infeasible. The goal is defined as a conjunction of  $(\text{AtPosition } o_{goal}, dir, o_{ref})$ . We give a reward of 3 for each  $\text{AtPosition}$  accomplished and -6 for sampling infeasible action. The discount factor  $\gamma$  is 0.99, and if we find a successful plan, we stop the planning and execute it.

## 4.2 Baselines

We compare STaLM with the following baselines:

- **UCT [47]:** Standard UCT without LLM's guidance.  $N_{budget}$  is set to 35.
- **UCT-with-Hcount [1]:** Instead of rollout, we use a modified version of a hand-designed heuristic for G-TAMP, Hcount, as a value function for UCT. Hcount estimates the cost-to-go based on the number of occlusions for  $o_{target}$ . Details are given in the Appendix D.  $N_{budget}$  is set to 35.

Initial config	Goal	Purpose & Challenge
	(Bottle 1, on, Table 1) (Bottle 2, on, Counter 2) (Bottle 3, on, Table 1) (Salter 1, on, Counter 2) (Salter 2, on, Shelf) (Salter 3, on, Table 2)	<ul style="list-style-type: none"> <li>The robot must open the door to place the bottle 2, salter 1 and 2</li> <li>Test if the robot understand occlusion by door</li> </ul>
	(Salter, on, Table 2) (Salter, right of, Bottle 1)	<ul style="list-style-type: none"> <li>Bottle 2 and 3 must be cleared before picking the salter</li> <li>The robot must know how to clear occlusion</li> <li>Test if the robot understand occlusion by movable obstacles</li> </ul>
	(Milk, on, Counter 2) (Coke, on, Counter 2)	<ul style="list-style-type: none"> <li>The robot must not randomly pick unrelated object (non-occluding bottles)</li> <li>The robot must not place the object in a position that occlude the milk or coke</li> <li>Test if the robot can pick object in tight space</li> </ul>
Initial config	Goal	Purpose & Challenge
	(Plate, on, Counter 1) (Salter, on, Table 2) (Salter, right of, Bottle 1)	<ul style="list-style-type: none"> <li>Bottle 2 and 3 must be cleared before picking the salter</li> <li>The robot must open the door before picking the plate</li> <li>Tests if the robot can prioritize actions unspecified by the goal.</li> </ul>
	(Sprite, on, Counter 1) (Coke, on, Counter 1) (BBQ grill, on, Counter 1)	<ul style="list-style-type: none"> <li>The robot must temporally place the BBQ grill to regions other than Counter 1 to make room for placing coke and sprite, and place the BBQ grill back to Counter 1</li> <li>Test if the robot can temporarily place the target to a region unspecified by the goal</li> </ul>
	(Coke, on, Counter 2) (Wine, on, Counter 2) (BBQ grill, on, Counter 2)	<ul style="list-style-type: none"> <li>When holding the grill, the robot must explore options other than placing it back on the counter 2.</li> <li>Test if the robot can temporarily displace the object already in the goal</li> </ul>

**Figure 4.** Description for problems showing the initial states and the goals. The purpose and possible challenges of each problem are listed. Goal objects are shown in cyan.

- *PDDLStream* [19]: a pure TAMP algorithm that uses a domain-independent heuristic function for guiding its search. We use an adaptive algorithm of PDDLStream.
- *SayCan* [14]: an LLM planner whose prompt includes task instruction and action history in the prompt. Because GPT does not support the LLM score evaluation, we use the empirical policy distribution from [9] with  $N_{batch} = 5$  responses as a likelihood of action and use action’s precondition as the affordance score.
- *Iterative-Replanning* [40]: uses our prompt to query the LLM for a single task plan and concretize it. If the plan does not succeed, Iterative-Replanning replans by appending up to two previously failed plans and calling the LLM for a new task plan. If no feasible plan is found after 5 attempts, the next action from the most recent plan is executed.
- *LLM-MCTS* [9]: computes the action distribution by counting the number of actions in the  $N_{budget} = 5$  LLM responses and using it for action selection with PUCT [46] in MCTS.  $N_{budget}$  is 35.

STaLM uses  $N_{batch} = 5, N_{budget} = 30$ . For all LLM-based methods, we use gpt-4-turbo-2024-04-09 with a decoding temperature of 1. Detailed hyperparameters of STaLM is included in Appendix C.

### 4.3 Results and analysis

Table 2 shows that STaLM outperforms all baselines in all problems in terms of planning speed and success rate. UCT performs poorly since it cannot simulate every possible placement for multiple steps ahead with  $N_{budget} = 35$ . UCT-with-Hcount calls the motion planner numerous times to compute the occlusion for every new state we encounter, leading to time-outs in *P1* and *P3*. STaLM, in contrast, computes occlusions only when querying the LLM. Furthermore, in *P5* and *P6*, UCT-with-Hcount fails to handle states in local optima where the objects already in the goal must be moved to another region because HCount explicitly penalizes such action. Such local optima are frequently encountered in *P5* when the tree search greedily follows the reward and is given as the initial state for *P6*.

PDDLStream treats each sample of continuous parameter as an “PDDL object”, an instance or item that exists in the world described by the PDDL domain and problem definitions, like a movable object or a region. Such PDDL objects are accumulated as the planning proceeds, slowing down the task planning [50]. For instance, in *P6*, PDDLstream fails to sample collision-free placement pose for the coke occluded by the BBQ grill, but it can still reuse the gripper pose or base trajectories if the occluders are cleared. Therefore, numerous gripper poses and base trajectories that have been tried are stored in a cache, often exceeding 2000 in number. This results in about 100 seconds to make a task plan because the task planner must consider the combinations of accumulated PDDL objects, eventually leading to time-out.

SayCan avoids infeasible action via affordance score but cannot prioritize occlusion-clearing action since the prompt lacks state information and the LLM score of clearing the obstacle is indistinguishable from other actions. In contrast, our prompt includes state information about occlusions, so the LLM provides the task plan that clears occlusions.

Iterative-replanning uses the LLM to modify a task plan based on the given feedback about what action of the task plan is infeasible. However for problems *P3-6*, where the set of feasible continuous parameters is relatively smaller than *P1-2*, a common failure mode was the LLM recklessly attributing the failure to the task plan even when the task plan could achieve success by further search of continuous parameters. STaLM, in contrast, is able to further search for continuous parameters of promising task plans by using WarmStartedUCT to initialize the tree with task plans from LLM and conducting MCTS.

All of LLM-MCTS leads to time-outs because, for each action, LLM-MCTS makes  $N_{budget}$  number of LLM calls to compute an empirical action distribution where each call takes 10-20 seconds. STaLM instead makes a single LLM query to compute multiple plans, requiring fewer LLM calls for the same number of simulations and saving a significant amount of time. Example LLM responses of STaLM is shown in Appendix E.

We examine the effectiveness of integrating tree search with LLM by comparing with a variant of STaLM, STaLM without UCT which is STaLM with  $N_{budget} = 0$ , and does not search further after Concretize. From Table 3, we see that STaLM without UCT shows lower performance than STaLM because it is limited to the LLM’s response. For

Method	Metric	P1	P2	P3	P4	P5	P6
UCT	Success rate	0.02	0.8	0.00	0.20	0.00	0.00
	Time (s)	229.43	93.892	<i>t/o</i>	122.48	<i>t/o</i>	<i>t/o</i>
UCT -with-Hcount	Success rate	0.00	0.90	0.00	0.36	0.08	0.00
	Time (s)	<i>t/o</i>	163.80	<i>t/o</i>	219.67	356.65	<i>t/o</i>
PDDLStream	Success rate	0.20	0.56	0.00	0.24	0.12	0.00
	Time (s)	148.88	107.06	<i>t/o</i>	130.27	232.97	<i>t/o</i>
SayCan	Success rate	0.02	0.14	0.00	0.00	0.00	0.00
	Time (s)	54.01	81.68	<i>t/o</i>	<i>t/o</i>	<i>t/o</i>	<i>t/o</i>
Iterative Replanning	Success rate	1.00	0.98	0.36	0.22	0.36	0.36
	Time (s)	64.00	94.47	260.76	163.43	163.42	128.83
LLM-MCTS	Success rate	0.00	0.00	0.00	0.00	0.00	0.00
	Time (s)	<i>t/o</i>	<i>t/o</i>	<i>t/o</i>	<i>t/o</i>	<i>t/o</i>	<i>t/o</i>
<b>STaLM</b> (ours)	Success rate	<b>1.00</b>	<b>1.00</b>	<b>0.84</b>	<b>0.96</b>	<b>0.88</b>	<b>0.96</b>
	Time (s)	31.22	54.18	241.67	148.37	233.21	165.85

**Table 2.** Success rate and average planning time of success for STaLM and baselines. Timeout (*t/o*) indicates cases where solutions were not found within the max time limit.

instance, in *P4*, the robot fails to open the door because it picks up the plate first and the hand is occupied. In *P5*, *P6*, the LLM is unaware that other objects cannot be placed in the goal region if the BBQ grill is placed first.

Computing occlusion predicates and asking LLM for challenges bring a non-trivial amount of increase in time to obtain task plans from LLM. So, to investigate the efficacy of our prompt design we test ablated versions of the prompt, as shown in Table 4. Without occlusion predicates, the LLM cannot perform state-based causal reasoning using geometric predicates, leading to the lowest performance of all ablated versions. Without asking for the challenge, the LLM returns responses that do not respect preconditions.

## References

- Kim B, Shimanuki L, Kaelbling LP et al. Representation, learning, and planning algorithms for geometric task and motion planning. *IJRR* 2021; .
- Garrett CR, Chitnis R, Holladay R et al. Integrated task and motion planning. *Annual review of control, robotics, and autonomous systems* 2021; .
- Driess D, Ha JS, Tedrake R et al. Learning geometric reasoning and control for long-horizon tasks from visual input. *ICRA* 2021; .
- Silver T, Chitnis R, Curtis A et al. Planning with learned object importance in large problem instances using graph neural networks. In *AAAI*.
- Driess D, Oguz O, Ha JS et al. Deep visual heuristics: Learning feasibility of mixed-integer programs for manipulation planning. In *ICRA*.
- Kim B, Kaelbling L and Lozano-Pérez T. Guiding search in continuous state-action spaces by learning an action sampler from off-target search experience. In *AAAI*, volume 32.
- Ortiz-Haro J, Ha JS, Driess D et al. Structured deep generative models for sampling on constraint manifolds in sequential manipulation. In *CoRL*. PMLR, pp. 213–223.
- Kambhampati S, Valmeekam K, Guan L et al. LLMs can’t plan, but can help planning in llm-modulo frameworks. *ICML* 2024; .
- Zhao Z, Lee WS and Hsu D. Large language models as commonsense knowledge for large-scale task planning. *NeurIPS* 2023; .
- Helmert M. The fast downward planning system. *JAIR* 2006; .
- Kim B and Shimanuki L. Learning value functions with relational state representations for guiding task-and-motion planning. *CoRL* 2019; .
- Kim B, Kaelbling LP and Lozano-Perez T. Learning to guide task and motion planning using score-space representation. *IJRR* 2017; .
- Skreta M, Yoshikawa N, Arellano-Rubach S et al. Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting. *ArXiv* 2023; .
- Ahn M, Brohan A, Brown N et al. Do as i can, not as i say: Grounding language in robotic affordances. *CoRL* 2022; .
- Silver T, Dan S, Srinivas K et al. Generalized planning in pddl domains with pretrained large language models. In *AAAI*.
- Lin K, Agia C, Migimatsu T et al. Text2motion: from natural language instructions to feasible plans. *Autonomous Robots* 2023; .
- Guan L, Valmeekam K, Sreedharan S et al. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *NeurIPS* 2023; .
- Xie Y, Yu C, Zhu T et al. Translating natural language to planning goals with large-language models. *ArXiv* 2023; .
- Garrett CR, Lozano-Pérez T and Kaelbling LP. Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning. In *ICAPS*.
- Ren T, Chalvatzaki G and Peters J. Extended tree search for robot task and motion planning. *ArXiv* 2021; .
- Couëtoux A, Hoock JB, Sokolovska N et al. Continuous upper confidence trees. In *LION*. Springer.
- Katz M, Sohrabi S and Udrea O. Top-quality planning: Finding practically useful sets of best plans. In *AAAI*.
- Toussaint M. Logic-geometric programming: An optimization-based approach to combined task and motion planning. In *IJCAI*.
- Ortiz-Haro J, Karpas E, Toussaint M et al. Conflict-directed diverse planning for logic-geometric programming. In *ICAPS*.
- Toussaint M and Lopes MC. Multi-bound tree search for logic-geometric programming in cooperative manipulation domains. *ICRA* 2017; .
- Braun CV, Ortiz-Haro J, Toussaint M et al. Rhh-lgp: Receding horizon and heuristics-based logic-geometric programming for task and motion planning. *IROS* 2021; .
- Silver D, Schrittwieser J, Simonyan K et al. Mastering the game of go without human knowledge. *Nature* 2017; .

Version	P1	P2	P3	P4	P5	P6
w/o UCT	1.00	1.00	0.78	0.76	0.82	0.94
STaLM	<b>1.00</b>	<b>1.00</b>	<b>0.84</b>	<b>0.96</b>	<b>0.88</b>	<b>0.96</b>

**Table 3.** Success rate of ablated version of STaLM

Prompting	P1	P2	P3	P4	P5	P6
Occ. info						
Ask Chall.						
-	1.00	0.92	0.64	0.72	0.28	0.32
-	0.98	0.84	0.64	0.50	0.20	0.20
✓	1.00	0.94	0.72	0.38	0.40	0.54
✓	<b>1.00</b>	<b>1.00</b>	<b>0.84</b>	<b>0.96</b>	<b>0.88</b>	<b>0.96</b>

**Table 4.** Success rate of ablated versions of prompt used by STaLM

28. Schrittwieser J, Antonoglou I, Hubert T et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature* 2019; .
29. Silver D, Huang A, Maddison CJ et al. Mastering the game of go with deep neural networks and tree search. *Nature* 2016; .
30. Driess D, Ha JS and Toussaint M. Deep visual reasoning: Learning to predict action sequences for task and motion planning from an initial scene image. *RSS* 2020; .
31. Khodeir MNM, Agro B and Shkurti F. Learning to search in task and motion planning with streams. *RA-L* 2021; .
32. Chitnis R, Silver T, Kim B et al. Camps: Learning context-specific abstractions for efficient planning in factored mdp. In *CoRL*.
33. Chitnis R, Kaelbling LP and Lozano-Perez T. Learning quickly to plan quickly using modular meta-learning. *ICRA* 2018; .
34. Kim B, Kaelbling L and Lozano-Pérez T. Adversarial actor-critic method for task and motion planning problems using planning experience. *AAAI* 2020; .
35. Silver T, Chitnis R, Tenenbaum JB et al. Learning symbolic operators for task and motion planning. *IROS* 2021; .
36. Silver T, Athalye A, Tenenbaum JB et al. Learning neuro-symbolic skills for bilevel planning. In *CoRL*.
37. Silver T, Chitnis R, Kumar N et al. Inventing relational state and action abstractions for effective and efficient bilevel planning. *AAAI* 2023; .
38. Kumar N, McClinton W, Chitnis R et al. Learning efficient abstract planning models that choose what to predict. In *CoRL*.
39. Li A and Silver T. Embodied active learning of relational state abstractions for bilevel planning. In *CoLLAs*.
40. Shinn N, Cassano F, Gopinath A et al. Reflexion: Language agents with verbal reinforcement learning. *NeurIPS* 2024; 36.
41. Rana K, Haviland J, Garg S et al. Sayplan: Grounding large language models using 3d scene graphs for scalable task planning. *CoRL* 2023; .
42. Chen Y, Arkin J, Zhang Y et al. Autotamp: Autoregressive task and motion planning with llms as translators and checkers. *ICRA* 2024; .
43. Singh I, Blukis V, Mousavian A et al. Progprompt: Generating situated robot task plans using large language models. *ICRA* 2023; .
44. Liu B, Jiang Y, Zhang X et al. Llm+p: empowering large language models with optimal planning proficiency. *ArXiv* 2023; .
45. Madaan A, Tandon N, Gupta P et al. Self-refine: Iterative refinement with self-feedback. *NeurIPS* 2023; .
46. Rosin CD. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence* 2011; .
47. Kocsis L and Szepesvári C. Bandit based monte-carlo planning. In *European conference on machine learning*. Springer.
48. Wei J, Wang X, Schuurmans D et al. Chain of thought prompting elicits reasoning in large language models. *NeurIPS* 2022; .
49. Coumans E and Bai Y. Pybullet, a python module for physics simulation for games, robotics and machine learning, 2016-2021.
50. Vu B, Migimatsu T and Bohg J. Coast: Constraints and streams for task and motion planning. *ICRA* 2024; .

## Appendix

### A Caching state computations

We cache grounded predicates (literals) to reduce redundant computations. During `PICK`, all the objects are not moved, so we reuse all the literals. For `OPEN`, the occlusion by the door is cleared, so we reuse all the literals except those about occlusions by the door. `PLACE` removes an object from one region and adds it to another. Therefore, we discard any literals by the moved object and recompute the occlusion for the objects in the target region only. We also save intermediate computations. We cache gripper pose sampled during `PICKOCCLUDEDBy` and reuse them to grasp the same target for `PLACEOCCLUDEDBy`. The robot path computed for each object without movable objects stays the same within the same state, so we cache all the collisions and reuse them in the same state.

### B Subroutines of STaLM

We provide the pseudocode for subroutines of STaLM described in Algorithm 1: `Concretize` that is used to concretizes the task plans and `WarmStartedUCT` which uses `WarmUpTree` to initialize the search tree with `Plans` and conducts MCTS with `Simulate`.

---

#### Algorithm 2: `Concretize(TaskPlans, s, h)`

---

```

1 Global Variables:  $T, H, \psi$ 
2  $\text{success} \leftarrow \text{False}$ ,  $\text{Plans} \leftarrow []$ 
3 for  $\text{TaskPlan}$  in  $\text{TaskPlans}$  do
4    $\text{plan} \leftarrow []$ 
5   for  $\mathbf{a}(\delta)$  in  $\text{TaskPlan}$  do
6      $\kappa \sim \psi(s, \mathbf{a}(\delta))$ ,  $s' \leftarrow T(s, \mathbf{a}(\delta, \kappa))$ 
7      $\text{plan.append}(\mathbf{a}(\delta, \kappa))$ 
8     if  $h > H$  or  $\mathbf{a}(\delta, \kappa)$  is not feasible then
9       break
10    if  $s' \in \mathcal{G}$  then
11       $\text{success} \leftarrow \text{True}$ 
12      break
13     $s \leftarrow s'$ ,  $h \leftarrow h + 1$ 
14   $\text{Plans.append}(\text{plan})$ 
15 return  $\text{success}$ ,  $\text{Plans}$ 

```

---

`Concretize` in Algorithm 2 takes as input `TaskPlans`, state  $s$ , and search depth  $h$ . For each discrete action  $\mathbf{a}(\delta)$  of `TaskPlan`, we sample continuous parameters  $\kappa$  with random sampler  $\psi$  and run the transition model (L5-6). If the resulting action  $\mathbf{a}(\delta, \kappa)$  is infeasible or exceeds max search depth  $H$ , the iteration stops for the plan (L8-9). If  $s'$  satisfies the goal  $\mathcal{G}$ , we terminate the process (L10-12). After iterating over all task plans, we return `success` and `Plans`(L15).

`WarmStartedUCT` shown in Algorithm 3 takes `Plans`, state  $s$ , search depth  $h$ , and planning budget  $N_{\text{budget}}$  as input. It starts by initializing the root node of tree  $\mathcal{T}$  with node value  $Q$ , number of visits  $n$ , and set of action parameters that have been tried  $U$  (L2). To consider the hybrid action space, search tree  $\mathcal{T}$  consists of two types of nodes: discrete node  $\mathcal{T}(s)$  where abstract choices are made and continuous node  $\mathcal{T}(s_{\mathbf{a}(\delta)})$  where continuous choices are made. Using



**Algorithm 3:** WarmStartedUCT(Plans,  $s, h, N_{budget}$ )

---

```

1 Global Variables:  $T, R, H$ 
2  $\mathcal{T}(s) = \{Q(s, \cdot) = 0, n = 0, U = \emptyset, \}$ 
3  $\mathcal{T} \leftarrow \text{WarmUpTree}(\mathcal{T}, s, \text{Plans})$ 
4 for  $i = 0$  to  $N_{budget}$  do
5   |  $\text{Simulate}(s, h, 0, \mathcal{T})$ 
6 return  $\mathcal{T}$ 

```

---

WarmUpTree, we warm-up the tree (L3) and perform UCT with  $N_{budget}$  times of Simulate (L4-5).

**Algorithm 4:** WarmUpTree( $\mathcal{T}, s, \text{ConcretePlans}$ )

---

```

1 Global Variables:  $T, R, H, \gamma$ 
2 for plan in Plans do
3   |  $\text{AddToTree}(\mathcal{T}, s, 0, \text{plan})$ 
4 return  $\mathcal{T}$ 
5 Function AddToTree( $\mathcal{T}, s, \text{total}, \text{plan}$ ):
6   |  $a(\delta, \kappa) = \text{plan.pop}()$ 
7   | if  $a(\delta) \notin \mathcal{T}(s).U$  then
8     |  $\mathcal{T}(s).U = \mathcal{T}(s).U \cup \{a(\delta)\}$ 
9     |  $\mathcal{T}(s_{a(\delta)}) = \{Q(s_{a(\delta)}, \cdot) = 0, n = 0, U = \emptyset\}$ 
10    |  $\mathcal{T}(s_{a(\delta)}).n = \mathcal{T}(s_{a(\delta)}).n + 1$ 
11    | if  $\kappa \notin \mathcal{T}(s_{a(\delta)}).U$  then
12      |  $\mathcal{T}(s_{a(\delta)}).U = \mathcal{T}(s_{a(\delta)}).U \cup \{\kappa\}$ 
13      |  $s' \leftarrow T(s, a(\delta, \kappa))$ 
14      |  $\mathcal{T}(s') = \{Q(s', \cdot) = 0, n = 0, U = \emptyset\}$ 
15      |  $s', r \leftarrow T(s, a(\delta, \kappa)), R(s, a(\delta, \kappa))$ 
16      |  $\mathcal{T}(s').n = \mathcal{T}(s').n + 1$ 
17      | if  $\text{len}(\text{plan})! = 0$  then
18        |  $\text{total} \leftarrow r + \gamma \text{AddToTree}(\mathcal{T}, s', \text{total}, \text{plan})$ 
19      | else
20        |  $\text{total} \leftarrow \text{Rollout}(s')$ 
21      |  $\mathcal{T}(s_{a(\delta)}).Q(s_{a(\delta)}, \kappa) += \frac{\text{total} - \mathcal{T}(s_{a(\delta)}).Q(s_{a(\delta)}, \kappa)}{\mathcal{T}(s_{a(\delta)}).n}$ 
22      |  $\mathcal{T}(s).Q(s, a(\delta)) += \frac{\text{total} - \mathcal{T}(s).Q(s, a(\delta))}{\mathcal{T}(s).n}$ 
23    | return total

```

---

WarmUpTree in Algorithm 4 takes input  $\mathcal{T}, s$ , and Plans. With AddToTree, Plans are used to initialize  $\mathcal{T}$ . AddToTree takes as input  $\mathcal{T}, s$ , accumulated reward  $total$  and plan. It pops  $a(\delta, \kappa)$  from a plan and initializes a continuous node  $\mathcal{T}(s_{a(\delta)})$  with the discrete action  $a(\delta)$  (L7-9). Similarly, it expands the continuous node with  $\kappa$  and initializes a discrete node (L11-15). AddToTree is recursively called up to the last action of a plan (L17). The value of leaf node is estimated by Rollout and backed-up (L19-22).

Simulate in Algorithm 5, takes as input  $s, h, total$  and  $\mathcal{T}$ . It begins by selecting a discrete action according to UCT, where exploration constant  $c$  is a hyperparameter, and adds a continuous node to the tree if the discrete action has not been tried (L3-5). Then, we use Progressive Widening (PW) [21] to sample a new  $\kappa$  only if the number of children in that node is below  $k_\alpha \cdot N(\tau)^{c_\alpha}$ , where  $k > 0$  and  $c_\alpha \in (0, 1)$  are hyperparameters. We select  $\kappa$  using UCT and run the transition and reward model with selected action  $\hat{a}(\delta, \kappa)$  to sample subsequent state  $s'$  (L7-10). This process is recursively iterated until the leaf node (L17-18) or simulation

**Algorithm 5:** Simulate( $s, h, total, \mathcal{T}$ )

---

```

1 Global Variables:  $T, R, \psi, H, \gamma, k_\alpha, c_\alpha$ 
  /* Select discrete action with UCT */
2  $\hat{a}(\delta) \leftarrow \arg \max_{a(\delta)} \mathcal{T}(s).Q(s, a(\delta)) +$ 
   $c \sqrt{\frac{\log \mathcal{T}(s).n}{1 + \mathcal{T}(s).child(a(\delta)).n}}$ 
3 if  $\hat{a}(\delta) \notin \mathcal{T}(s).U$  then
4   |  $\mathcal{T}(s).U = \mathcal{T}(s).U \cup \{\hat{a}(\delta)\}$ 
  /* Add continuous node to tree */
5   |  $\mathcal{T}(s_{\hat{a}(\delta)}) = \{Q(s_{\hat{a}(\delta)}, \cdot) = 0, n = 0, U = \emptyset\}$ 
6   |  $\mathcal{T}(s_{\hat{a}(\delta)}).n = \mathcal{T}(s_{\hat{a}(\delta)}).n + 1$ 
  /* Progressive Widening */
7   | if  $|\mathcal{T}(s_{\hat{a}(\delta)}).U| \leq k_\alpha (\mathcal{T}(s_{\hat{a}(\delta)}).n)^{c_\alpha}$  then
8     |  $\kappa \sim \psi(s, \hat{a}(\delta)), \mathcal{T}(s_{\hat{a}(\delta)}).U =$ 
9     |  $\mathcal{T}(s_{\hat{a}(\delta)}).U \cup \{\kappa\}$ 
  /* Select continuous parameter with UCT */
9   |  $\kappa \leftarrow \arg \max_{\kappa \in \mathcal{T}(s_{\hat{a}(\delta)}).U} \mathcal{T}(s_{\hat{a}(\delta)}).Q(s_{\hat{a}(\delta)}, \kappa) +$ 
   $c \sqrt{\frac{\log \mathcal{T}(s_{\hat{a}(\delta)}).n}{1 + \mathcal{T}(s_{\hat{a}(\delta)}).child(\kappa).n}}$ 
10  |  $s', r \leftarrow T(s, \hat{a}(\delta, \kappa)), R(s, \hat{a}(\delta, \kappa))$ 
  /* Add discrete node to tree */
11  | if  $\kappa \notin \mathcal{T}(s_{\hat{a}(\delta)}).U$  then
12    |  $\mathcal{T}(s') = \{Q(s', \cdot) = 0, n = 0, U = \emptyset\}$ 
13    |  $h \leftarrow h + 1, \mathcal{T}(s').n = \mathcal{T}(s').n + 1$ 
14    | if  $\mathcal{T}(s').U \neq \emptyset$  then
15      | if  $h > H$  or  $s' \neq \text{feasible}$  or  $s' \in \mathcal{G}$  then
16        |  $total \leftarrow r$ 
17      | else
18        |  $total \leftarrow r + \gamma \text{Simulate}(s', h, total, \mathcal{T})$ 
19    | else
20      |  $total \leftarrow \text{Rollout}(s')$ 
21    |  $\mathcal{T}(s_{\hat{a}(\delta)}).Q(s_{\hat{a}(\delta)}, \kappa) += \frac{total - \mathcal{T}(s_{\hat{a}(\delta)}).Q(s_{\hat{a}(\delta)}, \kappa)}{\mathcal{T}(s_{\hat{a}(\delta)}).n}$ 
22    |  $\mathcal{T}(s).Q(s, \hat{a}(\delta)) += \frac{total - \mathcal{T}(s).Q(s, \hat{a}(\delta))}{\mathcal{T}(s).n}$ 
23  | return total

```

---

is terminated due to reaching the max planning horizon  $H$ , encountering infeasible action, or achieving  $\mathcal{G}$  (L15-16). At the leaf node, the value of the node is estimated by Rollout and backed up along selected nodes (L19-22).

## C Hyperparameters of STaLM

For STaLM, we use  $(N_{batch}, N_{budget}) = (5, 30)$  and gpt-4-turbo-2024-04-09 with a decoding temperature of 1 to generate TaskPlans in QueryLLM. For WarmStartUCT of STaLM, We use UCT exploration constant  $c = 50$ , PW constants  $(k_\alpha, c_\alpha) = (1.5, 0.15)$ .

## D Hand designed heuristics

The modified Hcount [1] used in UCT-with-Hcount is:

$$Hcount = |M| - |O_{\text{InGoal}}| + \mathbb{1}_{O_{\text{target}} \in O_{\text{InGoal}}} - \mathbb{1}_{O_{\text{target}} \notin O_{\text{InGoal}} \wedge O_{\text{target}} \in O_{\text{goal}}}$$

$M$  is the set of objects that occludes the goal objects  $O_{\text{goal}}$  and that recursively occludes those occluders.  $O_{\text{InGoal}}$  refers

to the objects already in the goal. We add 1 if  $o_{target} \in O_{InGoal}$  and discount 1 if  $o_{target} \notin O_{InGoal}$  is an goal object. Since Hcount computes cost-to-go, we convert it to a value function for a node by using  $Q = 3 \cdot n(O_{goal}) - 3 \cdot Hcount$ , where  $Q$  is the node value.

## E Example of LLM response in STaLM

In the following, we provide examples of responses from LLM queries made in STaLM for *P1-6*.

### ## Possible Challenges for unachieved goals based on current state ##

1. Multiple objects are occluded by 'kitchen\_door', which makes picking them difficult unless 'kitchen\_door' is opened.
2. The locations of the objects 'salter1', 'salter2', 'bottle1', 'bottle2', and 'bottle3' need to be changed, but their placement may be occluded by various objects or by 'kitchen\_door'.

### ## Plan ##

```
plan = [ ('open', 'kitchen_door'), # Open the kitchen door
to prevent occlusion
('pick', 'bottle1'), # Pick bottle1 from table2
('place', 'bottle1', 'on', 'table1'), # Place bottle1 on table1
('pick', 'bottle2'), # Pick bottle2 from minifridge
('place', 'bottle2', 'on', 'counter2'), # Place bottle2 on
counter2
('pick', 'salter1'), # Pick salter1 from counter1
('place', 'salter1', 'on', 'counter2'), # Place salter1 on
counter2
('pick', 'salter2'), # Pick salter2 from counter2
('place', 'salter2', 'on', 'shelf'), # Place salter2 on shelf
('pick', 'bottle3'), # Pick bottle3 from table2
('place', 'bottle3', 'on', 'table1'), # Place bottle3 on table1
('pick', 'salter3'), # Pick salter3 from table1
('place', 'salter3', 'on', 'table2') # Place salter3 on table2
]
```

Figure 5. Example LLM response of STaLM in P1

### ## Possible Challenges for unachieved goals based on current state ##

1. \*\*Relocation of 'salter'\*\*: The salter is currently positioned on 'counter2' and needs to be relocated to 'table2' as per the goal. However, any pick action for the salter is occluded by both 'bottle2' and 'bottle3'.
2. \*\*Accessibility of 'salter'\*\*: Before 'salter' can be picked up, the occluding items ('bottle2' and 'bottle3') need to be moved.
3. \*\*Placement of 'salter' by 'bottle1' on 'table2'\*\*: The specified final position for the 'salter' is 'right\_of' 'bottle1' on 'table2'. Currently, 'bottle1' is already on 'table2'. So, the 'salter' needs to be placed adjacent to it correctly.

### ## Plan ##

```
plan = [('pick', 'bottle3'), # Pick up bottle3 which
occludes picking up the salter
('place', 'bottle3', 'left_of', 'bottle2'), # Temporarily
place it to clear the path
('pick', 'bottle2'), # Pick up bottle2 which also occludes
the salter
('place', 'bottle2', 'left_of', 'bottle3'), # Temporarily
place it so salter can be accessed
('pick', 'salter'), # Now pick up the salter
('place', 'salter', 'on', 'table2'), # Place the salter on
table2
('place', 'salter', 'right_of', 'bottle1'), # Finally, place the
salter right of bottle1 as required in the goal ]
```

Figure 6. Example LLM response of STaLM in P2

### ## Possible Challenges for unachieved goals based on current state ##

1. **Moving objects to different positions**: To achieve the goal of having milk and coke on 'counter2', these objects need to be picked up and placed on the counter.
2. **Obstructed Picks**: The initial state mentions occlusions in picking certain items including milk and coke, thus complicating the pick action.
3. **Unavailable Target Region**: 'counter2' is not declared in the problem's object definitions, assuming it's an error, and considering placement on 'counter1' instead.

### ## Plan ##

```
plan = [('pick', 'coke'),
('place', 'coke', 'on', 'counter1'),
('pick', 'milk'),
('place', 'milk', 'on', 'counter1')]
```

Figure 7. Example LLM response of STaLM in P3

### ## Possible Challenges for unachieved goals based on current state ##

1. **Obstacles and Occlusions**: Moving or placing objects will be a significant challenge due to multiple occlusions, as detailed in the problem.
2. **Opening the Kitchen Door**: The kitchen door is currently closed ('Closed kitchen\_door'), which is occluding several actions for moving objects. Thus, the door must be opened first to reduce occlusions and facilitate other actions.
3. **Handling Occluded Actions**: The picking up of 'salter' is currently occluded by both 'bottle2' and 'bottle3'. Similarly, placing 'plate' on 'counter1' has occlusions.
4. **Switch between picking and placing**: Robot will have to alternately pick and place objects, and this requires ensuring that the robot hand is available ('HandAvailable').

### ## Plan ##

```
plan = [ ('open', 'kitchen_door'), # Open the kitchen door
to remove several occlusions
('pick', 'plate'), # Pick up the plate, assuming the
kitchen_door being open has resolved the occlusion
('place', 'plate', 'on', 'counter1'), # Place the plate on
counter1
('pick', 'salter'), # Pick up the salter
('place', 'salter', 'on', 'table2'), # Place the salter on table2
('place', 'salter', 'right_of', 'bottle1') # Place the salter to
the right of bottle1 on table2]
```

Figure 8. Example LLM response of STaLM in P4

## F Example of prompt used in STaLM

In Figure 11, we provide an example of a prompt used to generate task plans with LLM in P5.

### ## Possible Challenges for Unachieved Goals Based on Current State ##

1. The 'coke' and 'wine' need to be placed on 'counter2' but their current positions and potential placements are occluded by the 'beef\_grill' and the 'kitchen\_door'.
2. 'PickOccludedBy' predicates indicate that 'coke' and 'beef\_grill' are occluded by the 'kitchen\_door', which is closed.
3. 'AtPosition' predicates show that all movable\_objects are initially located on different counters and tables than 'counter2', where the objective is to position them.

To achieve the goal, certain actions involving moving occluding objects or opening doors will be needed.

### ## Plan ##

```
plan = [('open', 'kitchen_door'), # Open the kitchen door
to ensure picking of coke and beef_grill is possible
('pick', 'coke'), # Pick coke, requirement: hand available
and not unsafe to pick
('place', 'coke', 'on', 'counter2'), # Place coke on
counter2, addressing one part of the goal
('pick', 'beef_grill'), # Pick beef_grill now that
kitchen_door is open
('place', 'beef_grill', 'on', 'counter2'), # Place beef_grill
on counter2, addressing another part of the goal
('pick', 'wine'), # Pick wine from its initial position
('place', 'wine', 'on', 'counter2') # Place wine on
counter2; achieving the final part of the goal]
```

Figure 9. Example LLM response of STaLM in P5

### ## Possible Challenges for Unachieved Goals Based on Current State ##

1. The 'coke' and 'wine' need to be placed on 'counter2' but their current positions and potential placements are occluded by the 'beef\_grill' and the 'kitchen\_door'.
2. 'PickOccludedBy' predicates indicate that 'coke' and 'beef\_grill' are occluded by the 'kitchen\_door', which is closed.
3. 'AtPosition' predicates show that all movable\_objects are initially located on different counters and tables than 'counter2', where the objective is to position them.

To achieve the goal, certain actions involving moving occluding objects or opening doors will be needed.

### ## Plan ##

```
plan = [('open', 'kitchen_door'), # Open the kitchen door
to ensure picking of coke and beef_grill is possible
('pick', 'coke'), # Pick coke, requirement: hand available
and not unsafe to pick
('place', 'coke', 'on', 'counter2'), # Place coke on
counter2, addressing one part of the goal
('pick', 'beef_grill'), # Pick beef_grill now that
kitchen_door is open
('place', 'beef_grill', 'on', 'counter2'), # Place beef_grill
on counter2, addressing another part of the goal
('pick', 'wine'), # Pick wine from its initial position
('place', 'wine', 'on', 'counter2') # Place wine on
counter2; achieving the final part of the goal ]
```

Figure 10. Example LLM response of STaLM in P6



## System prompt

You are an expert proficient in PDDL and planning actions for a problem. Your response should follow this template:  
## Possible Challenges for unachieved goals based on current state ##

## Plan for unachieved goals ## plan = [('action\_type', 'args\_1', 'args\_2',...), ]

## User prompt

### Domain ###

```
(define (domain shop)
(:requirements :typing)
(:types movable_object region openable - object )
(:constants on left_of right_of front_of behind_of - direction )
```

(:predicates

```
(RobotAt ?region) ; True if robot is at region
(RobotHolding ?movable_object) ; True if robot is holding
movable_object
(HandAvailable ) ; True if robot hand is available
(AtPosition ?subject ?direction ?reference) ; True if subject is
at direction of reference
(IsClosed ?door) ; True if the door is closed
(PickOccludedBy ?subject ?occluder) ; True is action (pick,
subject) is occluded by occluder
(PlaceOccludedBy ?subject ?direction ?reference ?occluder) ;
True if action (place, subject, direction, reference) is occluded
by occluder )
```

(:action pick ; example ('pick', 'bottle')

:parameters (?subject )

:precondition (and (HandAvailable ) (not (UnsafePick ?subject)))

:effect (and (not (HandAvailable )) (RobotHolding ?subject) (not (AtPosition ?subject ?direction ?reference))) )

(:action place ; example ('place', 'bottle', 'behind\_of', 'can')

:parameters (?subject ?direction ?reference)

:precondition (and (RobotHolding ?subject) (not (UnsafePlace ?subject ?direction ?reference)))

:effect (and (not (RobotHolding )) (HandAvailable ) (AtPosition ?subject ?direction ?reference)) )

(:action open ; example ('open', 'door')

:parameters (?subject )

:precondition (and (IsClosed ?subject) (HandAvailable ))

:effect (and (not (IsClosed ?subject))) )

(:derived (UnsafePick ?subject)

(exists (?occluder) ((PickOccludedBy ?subject ?occluder))))

(:derived (UnsafePlace ?subject ?direction ?reference)

(exists (?occluder) ((PlaceOccludedBy ?subject ?direction ?reference ?occluder)))) )

### Problem ###

```
(objects coke - movable_object sprite - movable_object
mango_juice - movable_object dr_pepper - movable_object
pork_grill - movable_object beef_grill - movable_object
kitchen_door - openable counter1 - region counter2 - region
counter3 - region counter5 - region )
```

(:init

```
[('HandAvailable', ''), ('AtPosition', 'coke', 'on',
'counter2'), ('AtPosition', 'beef_grill', 'on', 'counter2'),
('AtPosition', 'coke', 'behind_of', 'beef_grill'), ('AtPosition',
'beef_grill', 'front_of', 'coke'), ('AtPosition', 'sprite',
'on', 'counter3'), ('AtPosition', 'mango_juice', 'on',
'counter5'), ('AtPosition', 'dr_pepper', 'on', 'counter5'),
('AtPosition', 'mango_juice', 'behind_of', 'dr_pepper'),
('AtPosition', 'dr_pepper', 'front_of', 'mango_juice'), ('Pick-
OccludedBy', 'coke', 'beef_grill'), ('PlaceOccludedBy',
'beef_grill', 'left_of', 'coke', 'coke'), ('PlaceOccludedBy',
'beef_grill', 'left_of', 'sprite', 'sprite'), ('PlaceOcclud-
edBy', 'beef_grill', 'right_of', 'mango_juice', 'dr_pepper'),
('PlaceOccludedBy', 'beef_grill', 'right_of', 'mango_juice',
'mango_juice'), ('PlaceOccludedBy', 'beef_grill', 'left_of',
'dr_pepper', 'dr_pepper'), ('PlaceOccludedBy', 'sprite',
'on', 'counter2', 'beef_grill'), ('PlaceOccludedBy', 'sprite',
'left_of', 'coke', 'beef_grill'), ('PlaceOccludedBy', 'sprite',
'right_of', 'coke', 'coke'), ('PlaceOccludedBy', 'sprite',
'right_of', 'coke', 'beef_grill'), ('PlaceOccludedBy',
'sprite', 'behind_of', 'coke', 'beef_grill'), ('PlaceOc-
cludedBy', 'sprite', 'behind_of', 'beef_grill', 'coke'),
('PlaceOccludedBy', 'sprite', 'behind_of', 'beef_grill',
'beef_grill'), ('PlaceOccludedBy', 'sprite', 'behind_of',
'mango_juice', 'dr_pepper'), ('PlaceOccludedBy',
'mango_juice', 'left_of', 'coke', 'beef_grill'), ('PlaceOc-
cludedBy', 'mango_juice', 'right_of', 'coke', 'beef_grill'),
('PlaceOccludedBy', 'mango_juice', 'behind_of', 'coke',
'coke'), ('PlaceOccludedBy', 'mango_juice', 'behind_of',
'coke', 'beef_grill'), ('PlaceOccludedBy', 'mango_juice',
'behind_of', 'beef_grill', 'coke'), ('PlaceOccludedBy',
'mango_juice', 'behind_of', 'beef_grill', 'beef_grill'),
('PlaceOccludedBy', 'mango_juice', 'behind_of', 'sprite',
'sprite'), ('PlaceOccludedBy', 'mango_juice', 'front_of',
'dr_pepper', 'dr_pepper'), ('PlaceOccludedBy', 'dr_pepper',
'on', 'counter2', 'beef_grill'), ('PlaceOccludedBy',
'dr_pepper', 'left_of', 'coke', 'beef_grill'), ('PlaceOc-
cludedBy', 'dr_pepper', 'right_of', 'coke', 'beef_grill'),
('PlaceOccludedBy', 'dr_pepper', 'behind_of', 'coke',
'coke'), ('PlaceOccludedBy', 'dr_pepper', 'behind_of',
'coke', 'beef_grill'), ('PlaceOccludedBy', 'dr_pepper',
'behind_of', 'beef_grill', 'coke'), ('PlaceOccludedBy',
'dr_pepper', 'behind_of', 'beef_grill', 'beef_grill')]
```

(:goal (and (AtPosition coke on counter1) (AtPosition sprite on counter1) (AtPosition beef\_grill on counter1)))

Generate a plan to achieve the goals from init.

Figure 11. Example of prompt used in STaLM in P5