A Supplementary Material

This Supplementary Material section provides additional details and discussions to complement our main paper. In section B, we expand on the discussion of our approach. Section C presents specific results for the seven tasks of the ALFRED challenge as well as new tasks that can be solved in the environment. Section D provides a formal definition of Object-Oriented Partially Observable Markov Decision Processes (OO-POMDPs), while section E provides a detailed background for our planning approach. Finally, section F provides an in-depth explanation of our method for the ALFRED challenge.

B Further Discussion

In section 9, we touched on various aspects of our methodology. This section provides more detail on these areas. Our approach to the ALFRED challenge hinges on egocentric planning and a user-defined symbolic planning model. We refer to these models as intentional, as they are purposefully crafted by the user, integrating specific environmental objects and properties. Symbolic models function as a form of world model that facilitates planning. Some methods seek to learn a latent representation, but this often comes with a high sample complexity (HPBL23). Modular meta-learning can reduce the sample complexity (ALPK18, ORCC21, MvSE22). If cost is not a constraint, world models can become components of efficient strategies trained to solve problems in specific domains (SAH$^+$20).

However, with limited resources or data, generalizing to new situations can be challenging. Take an embodied agent in an environment (def D.8) with N skills and M types of objects, tasked with carrying out a sequence of tasks that require plans involving a new combination of k skills and m types of objects, where $n \ll N$ and $m \ll M$[3]. Such setting is within the scope of methods like meta-learning or continual learning for adapting the latent representation for such unfamiliar environments (FAL17). The priors in some modular meta-learning approaches are related with our notions of objects and actions, but their inference algorithms might not scale well for long plans or ignore that some users might also want intentional modelling (ALPK18, ORCC21, MvSE22). Egocentric planning, on the other hand, may provide predictable scalability in these scenarios. While the definition of open-world generalization might be more elusive, our method leans towards systematic generalization across actions and object types.

Planning using certain latent representations or world models can also prove costly. As explained by (AKFM22) (LATPLAN), not all factored representations afford the same level of scalability. The authors of LATPLAN apply variational autoencoders to learn a discrete latent representation, incorporating additional priors so the representation can be interpreted as a STRIPS symbolic model, the most basic type of symbolic model we use for egocentric planning. Symbolic models like STRIPS enable solvers with greater scalability and generalization.

An alternative approach involves methods that learn a symbolic planning model (AFP$^+$18, CDVA$^+$22). Object-oriented POMDPs are especially suitable as the grounding on objects and action labels allows for more efficient algorithms (CMW13, MCRW09), an advantage not enjoyed by the authors of LATPLAN as they only use pairs of images as training data. Learning a symbolic model helps manage the risk of compounding errors since the model can be debugged and improved. Real-world applications may require monitoring the agent's behaviour for specific objects or actions. Furthermore, specific actions may carry inherent risks that need to be avoided. In general, we anticipate that developers of embodied agents will use as much learning as possible, as much intentional modelling as necessary, and symbols to blend the two of them (KSV$^+$22).

Further integration between symbolic planning models and other ML models might help to reduce some limitations of our approach. For instance, in ALFRED, searching for a pan should make the agent head toward the kitchen. If another model like an LLM finds it likely that the pan is in the kitchen, we can use the scores to produce action cost in a similar way to the integration done in SayCan, (iBC$^+$23).

Egocentric planning connects to a body of work aimed at tackling expressive planning problems by harnessing the scalability of classical planners. While egocentric planning focuses on the partial observability of objects, (APG09, PG09) consider partial observability given a set of possible initial

[3] While the state space is unknown to the agent, it is aware of the actions and object types.

states and deterministic actions, and (MMB14) considers full observability but with nondeterministic actions. Though these strategies can provide complete solutions proven offline, they can also serve as approximate methods for solving online problems. When planning with probabilistic effects, replanning assuming randomized determinization of actions can be a strong approach if there are no dead ends (YFG07).

Our symbolic planning models are extensions of STRIPS, but other options exist. Hierarchical task networks (HTN) (NAI$^+$03, GA14), for example, offers an alternative to STRIPS as they allow the specification of plans at various abstraction levels. Sometimes, it is more practical to outline potential plans using formalisms like linear temporal logic (LTL) formulae (DV13). Nonetheless, the relationship between these models has been examined, and it is feasible to integrate them with STRIPS-based planning (CC04, HBB$^+$19, MH13).

C Results in ALFRED's Task and New Tasks

C.1 ALFRED Task Performance

ALFRED Tasks	Valid Unseen		Valid Seen	
	SR	GC	SR	GC
look_at_obj_in_light	53.77	54.27	47.22	30.11
pick_and_place_simple	43.14	47.11	37.15	39.77
pick_cool_then_place_in_recep	46.76	45.82	47.01	48.89
pick_clean_then_place_in_recep	38.77	41.37	38.01	42.21
pick_cool_then_place_in_recep	46.26	46.81	47.21	47.33
pick_heat_then_place_in_recep	37.69	40.50	39.82	41.33
pick_two_obj_and_place	30.33	35.71	27.37	31.21

Table 5: Performance on Valid Unseen and Valid Seen for each task

To analyze the strengths and weaknesses of our approach in different types of tasks, we conducted further evaluations by task type as in Table 5. Our findings reveal several noteworthy observations. Firstly, the success rates (SR) and goal completion rates (GC) for the "Pick_two_obj_and_place" task are lower than the rest, mostly due to these tasks involving more objects and with smaller masks. The task "look_at_obj_in_light" has the best performance due to the variety of lamp is low compare to other common objects. Secondly, there is no strong correlation between the number of interactions involved in a task type and its performance (STG$^+$20). For example, the task "pick_cool_then_place_in_receptacle" has an extra cooling step compare to "pick_and_place_simple" but still have better performance. A detailed description of each task can be found in the original ALFRED paper on page 16 to 19 (STG$^+$20).

C.2 New Task Performance

In section 8.4, we discuss the generalization of our approach to new tasks. The results are summarized in table 6.

	SR	GC	PLWSR	PLWGC
Pickup and Place Two Objects	90.00	93.44	3.71	4.37
Clean-and-Heat Objects	80.00	89.77	2.56	3.90
Clean-and-Cool Objects	85.00	93.38	4.44	4.78
Heat-and-Cool Objects	80.00	89.20	4.01	4.66
Pick and Place Object in Drawer	70.00	77.81	2.59	3.22

Table 6: New tasks

Our approach enable us to reuse the definition of existing models to achieve zero-shot generalization to new tasks as long as the set of actions, objects and their relationships remain the same. In many embodied agent applications, agent need to have ability to adapt to new tasks. For instance, organizing a living room could involve storing loose items on a shelf or arranging them on a table. An

agent familiar with common household objects and equipped with the skills to handle them should seamlessly manage both types of cleaning tasks. However, transfer learning for neural networks remains challenging in the context of embodied agent tasks generalizing across objects and skills. Similarly, the template system used in FILM demands hand-crafted policies for new task types. The ALFRED dataset include instructions for each tasks that (STG+20), although our approach does not use them. In a our egocentric planning formulation, a task can be broken down into a set of goal conditions that comprise objects of interest. The planner can then autonomously generate an action sequence to reach these goals. Consequently, our method can independently execute these tasks without the need for transfer learning or the addition of new task templates. To demonstrate this, we devised 5 new task types that were not present in the training data. We have chosen environments in which all objects for each task have been successfully manipulated in at least one existing ALFRED task. This selection is made to minimize perception errors. These tasks share the same world model as ALFRED and thus do not necessitate any additional modification beyond specifying a new goal. We selected environments and layouts from the training set where subgoals required for each new task type are feasible, ensuring the feasibility of the new task for our agent. Table 6 lists these new tasks and their success rates. We selected 20 scenarios for each new task and maintained the maximum 10 interaction error as in the ALFRED problem. Our method was able to achieve an impressive overall success rate of 82% without any additional adjustments beyond the new goal specification. However, due to the constraints of the ALFRED simulator, we're unable to experiment with introducing new objects and action types beyond what's available. For instance, "heating" in ALFRED is solely accomplished via a microwave. In real-world scenarios, we could easily incorporate ovens and actions to operate them with minor alterations to our PDDL domains. Similarly, we could introduce time and resource restrictions for each action, which are common in many PDDL benchmarks. The ability to accommodate new goals and constraints is a key advantage of our planning-based approach.

C.3 Computation

Perception and Language module was fine-tuned on a Nvidia 3080. The planning algorithm uses a 8 core Intel i7 CPU with 32 GB of ram. More detailed on traning can be found in (MCR+22)

D Embodied Agents Generalizing within given Objects Types

In this section, we provide further details about embodied agents presented in section 4. Embodied agents perceive and act in environments to accomplish tasks. Navigating challenges and constraints imposed by their environment and tasks, these agents ensure safety and adhere to physical restrictions. Taskable agents rely on their surroundings and inherent capabilities, creating a strong dependency and often specializing in particular environments. This specialization shapes their cognition and problem-solving strategies, presenting an opportunity for robust agent design.

We exploit this limitation by focusing on generalization across tasks involving the same object types and their relationships, applicable to both physical and software agents. Taskable agents possess specialized capabilities for these object types, as seen in robotics where battery level, body state, and actuators are distinct variables (CST+22). Object types also serve as inputs/outputs in end-to-end approaches with learned representations and appear in embodied agents operating in software environments, using APIs and maintaining internal states. Our symbolic planning approach aims to provide a basis for designing adaptable and flexible agents that sense and act upon given object types. We start by defining object types, and entities grounded in these types.

Definition D.1 (Objects and Object Types)**.** An *Object Type*, denoted as $\langle \mathcal{T}, \mathcal{V} \rangle$, consist of a finite set of object types $\mathcal{T}$, along with a mapping $\mathcal{V} : t_i \to \mathcal{V}_i$ that assigns each type $t_i \in \mathcal{T}$ a set of possible values $\mathcal{V}_i$. An *Object* $o = (t_i, v)$ is an *instance* of a type with a specific value, i.e., $t_i \in \mathcal{T}$ and $v \in \mathcal{V}(t_i)$.

Example D.2 (Object types and Values)**.** *An example of an object type is $t_{objectSeen}$ with values $\mathcal{V}_{(x,y)} \times \mathcal{V}_{direction} \times \mathcal{V}_{objectSubtype} \times \mathcal{V}_{id}$ where $\mathcal{V}_{(x,y)}$ is position in a grid, $\mathcal{V}_{direction}$ is the direction the object is facing, $\mathcal{V}_{objectSubtype}$ is the category of the object observed, and $\mathcal{V}_{id}$ is a unique identifier for the object. Other related examples of object types are $t_{movable}$ with values $\mathcal{V}_{movableSubtype} \times \mathcal{V}_{id} \times \mathcal{V}_{objectproperties}$, and $t_{obstacle}$ with values $\mathcal{V}_{obstacleSubtype} \times \mathcal{V}_{id} \times \mathcal{V}_{objectproperties}$. Furthermore, an action `pickup` with parameters $\Phi_{\mathcal{E}}(a) = (t_{objectSeen})$ is an action that picks up a specific object.*

Object types are used as parameters of entities of actions of Object-oriented POMDP defined, and as parameters of actions and predicates in planning domains in section E.

Definition D.3 (Typed Entity)**.** Given a set of entities Ξ, object types $\langle \mathcal{T}, \mathcal{V} \rangle$, and a mapping $\Phi : \xi \to \bigcup_{k \in \mathbb{N}} \mathcal{T}^k$. A *Type Entity* $\xi \in \Xi$ is mapped to $\Phi(\xi) = (t_1, t_2, ..., t_{k_\xi})$, with each $t_i \in \mathcal{T}$. This mapping associates each entity with a k_ξ-tuple of object types, corresponding to k_ξ parameters of the entity, defining the *arity* of the entity ξ. We refer to the variables $var \in \{var_1, var_2, ..., var_{k_\xi}\}$ as the positional parameters of the entity k_ξ, denoted $\xi(var_1, var_2, ..., var_k)$.

Definition D.4 (Grounded Entity)**.** Given a set of typed entities Ξ, with object types $\langle \mathcal{T}, \mathcal{V} \rangle$, and a mapping Φ, a *Grounded Entity*, denoted as ξ^g for $\xi \in \Xi$, is a parametric object where each type argument of ξ specified by $\Phi(\xi)$ is replaced by an object of the corresponding type. Specifically, $\xi^g = \xi(o_1, o_2, ..., o_{k_\xi})$, where each $o_i = (t_i, v_i)$ with $v_i \in \mathcal{V}(t_i)$. The set of all grounded entities is denoted by Ξ^g.

Now we define the environment where the embodied agent operates as a Partially Observable Markov Decision Process (POMDP) with costs. Cost POMDPs are related to classical MDPs but feature partial observability and use costs instead of rewards (MK12, GB13).

The literature sometimes confound the environment with the underlying the POMDP, but we want to make explicit what is available to the agent. Thus, while our underlying model is a POMDP, we consider agents that can aim to generalize without being fully aware of the distribution of future tasks, except for the actions and object types that are explicit in the environment. Therefore, we abuse slightly the notation to annotate with the objects defined below with the environment, denoted $\mathcal{E}$.

Definition D.5 (Cost POMDP)**.** A *Cost POMDP* is a tuple $\langle S_\mathcal{E}, A_\mathcal{E}, T_\mathcal{E}, C_\mathcal{E}, P_\mathcal{E}, Z_\mathcal{E} \rangle$, where $S_\mathcal{E}$ is a set of states, $A_\mathcal{E}$ is a finite set of actions, $T_\mathcal{E} : S_\mathcal{E} \times A_\mathcal{E} \times S_\mathcal{E} \to [0, 1]$ is the transition function, $C_\mathcal{E} : S_\mathcal{E} \times A_\mathcal{E} \to \mathbb{R} > 0$ is the cost function, $P_\mathcal{E}$ is a set of perceptions, and $Z_\mathcal{E} : S_\mathcal{E} \times A_\mathcal{E} \times P_\mathcal{E} \times S_\mathcal{E} \to [0, 1]$ is the sensor function.

Object-oriented POMDP is an extension of a Cost POMDP that includes objects and their properties to be used for expressing perceptions and actions.

Definition D.6 (Object-oriented POMDP)**.** An *Object-oriented POMDP* Γ, represented by $\langle S_\mathcal{E}, A_\mathcal{E}, \Phi_\mathcal{E}, T_\mathcal{E}, C_\mathcal{E}, P_\mathcal{E}, Z_\mathcal{E}, \mathcal{T}_\mathcal{E}, \mathcal{V}_\mathcal{E} \rangle$, extends a Cost POMDP by incorporating a predefined set of *Object Types* $\langle \mathcal{T}_\mathcal{E}, \mathcal{V}_\mathcal{E} \rangle$. It defines the set of perceptions $P_\mathcal{E}$ where a perception $p \in P_\mathcal{E}$ is a finite set of objects $\{o_1, o_2, \ldots, o_m\}$, and the set of typed actions $A_\mathcal{E}$ defined for the object types $\langle \mathcal{T}_\mathcal{E}, \mathcal{V}_\mathcal{E} \rangle$ and the map $\Phi_\mathcal{E}$.

In this context, *Grounded Actions* refer to grounded entities, where the set of entities is the actions $A_\mathcal{E}$. Values in object-oriented POMDPs can be dense signals like images or sparse signals like object properties. Values can have any structure, so they can refer to values in the domain of other types. Indeed, Object-oriented POMDPs can express Relational POMDPs (WK10, WJK08, SB09, TGD04, WPY05, DNR+20). However, our approach emphasizes the compositionality of the objects and their properties.

Agents interact with the POMDP through an environment. Given a task, we initialize the environment using the task details. The agent receives the tasks and acts until it achieves the goal or fails due to some constraint on the environment. The agent achieves a goal if, after a sequence of actions, the environment's hidden state satisfies the goal.

Definition D.7 (Task)**.** Given an Object-oriented POMDP Γ, a *Task* is a tuple $T_\mathcal{E} = \langle I_\mathcal{E}, G_\mathcal{E} \rangle$, where $I_\mathcal{E}$ is an object that the environment interpret for setting the initial state, and $G_\mathcal{E}$ is a set of goal conditions expressed as objects $\{g_1, g_2, \ldots, g_n\}$, with each g_i being an instance of an object.

Definition D.8 (Object-oriented Environment)**.** Given a hidden Object-oriented POMDP Γ, an *Object-oriented Environment*, or just *Environment*, is a tuple $\mathcal{E} = \langle A_\mathcal{E}, \mathcal{T}_\mathcal{E}, \mathcal{V}_\mathcal{E}, \mathsf{reset}, \mathsf{step} \rangle$, where $A_\mathcal{E}$, $\mathcal{T}_\mathcal{E}$, and $\mathcal{V}_\mathcal{E}$ are as previously defined, $\mathsf{reset} : T_\mathcal{E} \to P_\mathcal{E}$ is a function that takes a task $T_\mathcal{E} = \langle I_\mathcal{E}, G_\mathcal{E} \rangle$, resets the environment's internal state using $I_\mathcal{E}$ and returns an initial observation, and $\mathsf{step} : A^g_\mathcal{E} \to P_\mathcal{E} \times \mathbb{R}$ is the interaction function that updates the environment's internal state according to the execution of a grounded action a_g, for $a \in A_\mathcal{E}$, and returns an observation and a cost.

In some environments, $I_\mathcal{E}$ might be opaque for the agent. In others, it might containg information not used by the environment like user's preferences about how to complete a task. However, we abuse this notion and refer to $I_\mathcal{E}$ as the initial state of the environment.

Separating Object-oriented POMDPs from environments allows us to consider the scope of the generalization of our agents within the same POMDP. While the agent cannot see the state space directly, it is aware of the actions and the object types.

Definition D.9 (Embodied Agent)**.** Given an environment $\mathcal{E} = \langle A_\mathcal{E}, \mathcal{T}_\mathcal{E}, \mathcal{V}_\mathcal{E}, \mathsf{reset}, \mathsf{step}\rangle$, and a task $T_\mathcal{E} = \langle I_\mathcal{E}, G_\mathcal{E}\rangle$, an *Embodied Agent* is a tuple $\langle M_I, \pi_M, M_U, G_M\rangle$, where $M_I : P_\mathcal{E} \times I_\mathcal{E} \times G_\mathcal{E} \rightarrow \mathcal{M}$ is a function that takes an initial observation, a task, and returns the initial internal state of the agent, $\pi_M : \mathcal{M} \rightarrow A^g_\mathcal{E}$ is the agent's policy function that maps mental states to a grounded action in $A^g_\mathcal{E}$, $M_U : \mathcal{M} \times A^g_\mathcal{E} \times P_\mathcal{E} \times \mathbb{R} \rightarrow \mathcal{M}$ is the mental state update function receiving a grounded executed action, the resulting observation, and the cost, and $G_M : \mathcal{M} \rightarrow [0, 1]$ is the goal achievement belief function that returns the belief that the agent has achieved the goal based on its mental state M.

Algorithm 2 provides a general framework for an agent's execution process. By instantiating the policy, mental state update function, and goal achievement belief function using appropriate techniques, the agent can adapt to a wide range of environments and tasks. Although new tasks can set the agent in unseen regions of the state space $S_\mathcal{E}$, the agent remains grounded on the known types and the actions parameterized on object types, $\mathcal{T}_\mathcal{E}$. They offer opportunities for generalization by narrowing down the space of possible policies and enabling compositionality. The agent's mental state M can include its belief state, internal knowledge, or other structures used to make decisions. The policy π_M can be a deterministic or stochastic function, learned from data or user-specified, or a combination of both. As the agent has an internal state, its policy can rely on a compact state or on the history of interactions.

Algorithm 2 Agent Execution

Input: Environment $\langle A_\mathcal{E}, \mathcal{T}_\mathcal{E}, \mathcal{V}_\mathcal{E}, \mathsf{reset}, \mathsf{step}\rangle$
Input: Agent $\langle M_I, \pi_M, M_U, G_M\rangle$
Input: Task $\langle I_\mathcal{E}, G_\mathcal{E}\rangle$
Input: Probability threshold θ
Output: Successful trace τ or Failure

1: $p \leftarrow \mathsf{reset}(I_\mathcal{E}, G_\mathcal{E})$
2: $\tau = []$ ▷ Empty trace
3: $M \leftarrow M_I(p, I_\mathcal{E}, G_\mathcal{E})$
4: **while** $G_M(M) < \theta$ **do**
5: $\quad a^g = a(o_1, o_2, ..., o_{k_a}) \sim \pi_M(M, A^g_\mathcal{E})$ ▷ where a in $A_\mathcal{E}$ and o_i are objects.
6: $\quad \tau$.append(a^g)
7: $\quad p', c \leftarrow \mathsf{step}(a^g)$
8: $\quad$ **if** $\mathit{failed} \in p'$ **then**
9: $\quad\quad$ **return** Failure
10: $\quad M \leftarrow M_U(M, a^g, p', c)$
11: **return** τ

The agent starts in the initial state set from $I_\mathcal{E}$ and aims to achieve the goal $G_\mathcal{E}$. While the agent does not know the true state of the environment, agents can aim to generalize across actions and the object types as they are always available. While we prefer policies with lower expected cost, our main concern is achieving the goal. It is possible that a task might be unsolvable, possibly because of actions taken by the agent. We assume that failures can be detected, for instance, by receiving a special observation type failed. We further assume that after failure, costs are infinite, and failed is always observed.

The Iterative Exploration Replanning (IER), Algorithm 1 in section 5, is a instance of algorithm 2. Instead of using a parameter θ, it considers the problem solved when it obtains a plan that achieves the goal.

E Detailed Background: Parameterized Full-Observable Symbolic Planning

In this section, we describe the background definitions of parameterized full-observable symbolic models mentioned in section 5.1. We define planning domains and problems assuming full observability, including classical planning that further assumes deterministic actions, also known as STRIPS (GNT04, GB13). Planning domains and problems rely on typed objects (Def D.1), as object-oriented POMDPs, but planning actions include a model of the preconditions and effects.

Definition E.1 (Parametric Full-Observable Planning Domain)**.** A *parametric full-observable planning domain* is a tuple $\mathcal{PD} = \langle \mathcal{T}, \mathcal{V}, \mathcal{P}, \mathcal{A}, \Phi\rangle$, where $\langle \mathcal{T}, \mathcal{V}\rangle$ is a set of object types, $\mathcal{P}$ is a set of typed predicates and $\mathcal{A}$ is a set of typed actions, both typed by the object types and the map Φ. Each action a has an associated $\textsc{Pre}_a$, expressing the preconditions of the action, and $\textsc{Eff}_a$, expressing its effects. For each grounded action $a(o_1, o_2, ..., o_k) \in A^g$, the applicability of the grounded action in a state s is determined by checking whether the precondition $\textsc{Pre}_a$ is satisfied in s. The resulting state after applying the grounded action is obtained by updating s according to the effect update $\textsc{Eff}_a$. The specification of $\textsc{Eff}_a$ is described below.

While the values in both parametric full-observable planning domains and object-oriented POMDPs can be infinite (def D.6), the values used in concrete planning problems are assumed to be finite. As a consequence, the set of grounded actions of a planning problems is finite (def D.4).

Definition E.2 (Parametric Full-Observable Planning Problem)**.** A *parametric full-observable planning problem* is a tuple $\langle \mathcal{PD}, \mathcal{O}, I, G \rangle$, where $\mathcal{PD} = \langle \mathcal{T}, \mathcal{V}, \mathcal{P}, \mathcal{A}, \Phi \rangle$ is a parametric full-observable planning domain, $\mathcal{O}$ is a finite set of objects, each associated with a type in $\mathcal{T}$, I is a description of the initial state specifying the truth value of grounded predicates in $\mathcal{P}^g$, G is a description of the goal specifying the desired truth values of grounded predicates in $\mathcal{P}^g$, where $\mathcal{P}^g$ are the predicates $\mathcal{P}$ grounded on object types $\langle \mathcal{T}, \mathcal{V} \rangle$ and the map Φ encoding their typed arguments.

This definition covers a wide range of symbolic planning problems featuring full observability including probabilistic effects (KHW95). In particular, the definition can be specialized for classical planning problems, where the actions are deterministic, corresponding to STRIPS, to the most studied symbolic model (GB13).

For classical planning, given an action $a \in \mathcal{A}$ parameterized by a list of object types $a(t_1, t_2, ..., t_k)$, their precondition or effect are formulas γ expresse in terms of the variables $var \in \{var_1, var_2, ..., var_k\}$ that refer to positional parameters of the action. We denote the grounded formula $\gamma(o_1, o_2, \ldots, o_k)$ as γ with each variable var_i that occurs in γ is replaced by corresponding objects o_i.

Definition E.3 (Parametric Classical Planning Domain)**.** A *parametric classical planning domain* is a parametric full-observable planning domain where each action $a \in \mathcal{A}$ has preconditions $\textsc{Pre}_a$, and an effect update $\textsc{Eff}_a$ represented as a tuple $(\textsc{Add}_a, \textsc{Del}_a)$ of add and delete effects, all sets of predicates in $\mathcal{P}$. Given a grounded action $a(o_{arg})$ with $o_{arg} = o_1, o_2, \ldots, o_k$, the precondition $\textsc{Pre}_a$ is satisfied in a state s if $\textsc{Pre}_a(o_{arg}) \subseteq s$. The effect update $\textsc{Eff}_a$ describes the resulting state as $(s \setminus \textsc{Del}_a(o_{arg})) \cup \textsc{Add}_a(o_{arg})$ after applying the action.

Definition E.4 (Parametric Classical Planning Problem)**.** Given a parametric classical planning domain $\mathcal{PD}$, a *parametric classical planning problem* is a parametric full-observable planning problem where I and G are conjunctions of predicates in $\mathcal{P}$ (equivalently viewed as sets).

As Egocentric Planning relies on replanning over a sequence of related planning problems, it is convenient to define some operations required for exploration, as the agent discovers more objects. They allow us to implement the function *ExploreAct* in Algorithm 1.

Definition E.5 (Precondition Extension)**.** Given a parametric full-observable planning problem and an action $a \in \mathcal{A}$, *extending the precondition* $\textsc{Pre}_a$ with $\Delta\textsc{Pre}_a$, denoted $\textsc{Pre}_a \oplus \Delta\textsc{Pre}_a$, is a new precondition that is satisfied in a state s if $\textsc{Pre}_a$ and $\Delta\textsc{Pre}_a$ are both satisfied in s.

Definition E.6 (Effect Extension)**.** Given a parametric full-observable planning problem and an action $a \in \mathcal{A}$, *extending the effect* $\textsc{Eff}_a$ with $(\Delta\textsc{Add}_a, \Delta\textsc{Del}_a)$, denoted $\textsc{Eff}_a \oplus (\Delta\textsc{Add}_a, \Delta\textsc{Del}_a)$, is a new effect that applied to a state s results the new state s' that is like s but with modifications $\textsc{Eff}_a$, then $\Delta\textsc{Del}_a$ is not true in s', and $\Delta\textsc{Add}_a$ is true in s', in that order, assuming that neither $\Delta\textsc{Del}_a$ nor $\Delta\textsc{Add}_a$ appear in $\textsc{Eff}_a$.

With these operations we can define the Egocentric Planning algorithm for parametric full-observable planning problems. For instance, in the case of probabilistic planning the preconditions can be defined as in classical planning, so the precondition update is the same. The effect update of probabilistic actions is a probability distribution over the possible effects. However, extending the effect with a new effect is straightforward as the new effect must enforce that $Pr(\textsc{Add}_a|s) = 1$ and $Pr(\textsc{Del}_a|s) = 0$.

For classical planning, the precondition extension is the union of the original precondition and the new precondition as both are a set of predicates. The $\textsc{Eff}_a$ of classical planning actions $(\textsc{Add}_a, \textsc{Del}_a)$ is a tuple of sets of predicates, so updating the effect amounts to updating both sets of predicates.

Definition E.7 (Precondition Extension Classical)**.** For parametric classical planning problems, *extending the precondition* $\textsc{Pre}_a \oplus \Delta\textsc{Pre}_a$ is $\textsc{Pre}_a \cup \Delta\textsc{Pre}_a$.

Definition E.8 (Effect Update Classical)**.** For parametric classical planning problems, an *update to the effect* $(\textsc{Add}_a, \textsc{Del}_a)$ with $(\Delta\textsc{Add}_a, \Delta\textsc{Del}_a)$ is $(\textsc{Add}_a \cup \Delta\textsc{Add}_a, \textsc{Del}_a \cup \Delta\textsc{Del}_a)$.

For ALFRED, as the planning domain in classical, *ExploreAct* (a, o) creates a copy of a extended with a new precondition `(unknown o)` so a can only be applied if o is still unknown. The effects are extended by deleting `(unknown o)` and adding `(explored)`, so applying the exploration action satisfies the goal `(explored)`.

F Approach: Additional Details

In this section, we provide additional details on our approach discussed in section 6. Both the environment and the symbolic planning model consider objects and parametric actions, but they are represented differently.

For instance, our approach involves a vision model that perceives in a scene objects of known classes, such as `apple` or `fridge`, including their depths and locations (see sections 4 and D). Similarly, the planning domain includes corresponding object types, such as `AppleType` and `FridgeType` (see sections 5.1 and E). When the agent perceives an apple at location `l1`, the algorithm creates a fresh object `o1` for the apple, sets its type to `(objectType o1 AppleType)`, and its location to `(objectAtLocation o1 l1)`. Additionally, the domain contains information about what can be done with the apple, such as `(canCool FridgeType AppleType)`. Likewise, while the ALFRED environment contains actions like `toggle`, the planning domain contains actions like `coolObject`, which is parameterized on arguments such as the location, the concrete object to be cooled, and the concrete fridge object. Finally, if the task provided in natural language requests to cool down an apple, the planning problem goal contains, for instance, `(exists (isCooled ?o))`, which is satisfied when some object has been cooled down.

The remainder of this section is structured as follows. First, we provide additional details on the Language and Vision modules, including the specific classes they detect and how they are utilized. Next, we elaborate on the semantic spatial graph and its role in generating the agent's mental model, which tracks object locations as the agent perceives and modifies the environment. We then provide specific details on the planning domain used for ALFRED, which completes the elements used in our architecture depicted in Figure 1. Finally, we conclude the section with additional details on the egocentric planning agent that won the ALFRED challenge.

F.1 Language and Vision Module

Table 7: List of Objects

alarmclock	apple	armchair	baseballbat	basketball
bathtub	bathtubbasin	bed	blinds	book
boots	bowl	box	bread	butterknife
cabinet	candle	cart	cd	cellphone
chair	cloth	coffeemachine	countertop	creditcard
cup	curtains	desk	desklamp	dishsponge
drawer	dresser	egg	floorlamp	footstool
fork	fridge	garbagecan	glassbottle	handtowel
handtowelholder	houseplant	kettle	keychain	knife
ladle	laptop	laundryhamper	laundryhamperlid	lettuce
lightswitch	microwave	mirror	mug	newspaper
ottoman	painting	pan	papertowel	papertowelroll
pen	pencil	peppershaker	pillow	plate
plunger	poster	pot	potato	remotecontrol
safe	saltshaker	scrubbrush	shelf	showerdoor
showerglass	sink	sinkbasin	soapbar	soapbottle
sofa	spatula	spoon	spraybottle	statue
stoveburner	stoveknob	diningtable	coffeetable	sidetableteddybear
television	tennisracket	tissuebox	toaster	toilet
toiletpaper	toiletpaperhanger	toiletpaperroll	tomato	towel
towelholder	tvstand	vase	watch	wateringcan
window	winebottle			

F.1.1 Language Module

We used a BERT-based language models to convert a structured sentence into goal conditions for PDDL (DCLT19). Since the task types and object types of defined in the ALFRED metadata, we use a model to classify the type of task given a high-level language task description listed in Table 9

Table 8: List of Receptacles

Bathtub	Bowl	Cup	Drawer
Mug	Plate	Shelf	Sink
Box	Cabinet	CoffeeMachine	CounterTop
Fridge	GarbageCan	HandTowelHolder	Microwave
PaintingHanger	Pan	Pot	StoveBurner
DiningTable	CoffeeTable	SideTable	ToiletPaperHanger
TowelHolder	Safe	BathtubBasin	ArmChair
Toilet	Sofa	Ottoman	Dresser
LaundryHamper	Desk	Bed	Cart
TVStand	Toaster		

Table 9: List of Goals

pick_and_place_simple	pick_two_obj_and_place
look_at_obj_in_light	pick_clean_then_place_in_recep
pick_heat_then_place_in_recep	pick_cool_then_place_in_recep
pick_and_place_with_movable_recep	

and a separate model for deciding objects and their state based on the output of the second model in Table 7 and Table 8. For example, given the task description of "put an apple on the kitchen table," the first model will tell the agent it needs to produce a "pick-up-and-place" task. The second model will then predict the objects and receptacle we need to manipulate. Then the goal condition will be extracted by Tarski and turn in to a PDDL goal like *(on apple table)* using the original task description and output of the both models. We use the pre-trained model provided by FILM and convert the template-based result into PDDL goal conditions suitable for our planner. Although ALFRED provides step-by-step instructions, we only use the top-level task description since they are sufficient to predicate our goal condition. Also, our agent needs to act based on the policy planner, which makes step-by-step instructions unnecessary. This adjustment could also make the agent more autonomous since step-by-step instructions are often not available in many embodied agent settings.

F.1.2 Vision Module

The vision module is used to ground relevant object information into symbols that can be processed by our planning module. At each time step, the module receives an egocentric image of the environment. The image is then processed into a depth map using a U-Net, and object maks using Mask-RCNN. (HGDG17, LCQ$^+$18). Both masks are WxH matrices where W, and N are the width and height of the input image. Each point in the depth mask represents the predicted pixel distance between the agent and the scene. The average depth of each object is calculated using the average of the sum of the point-wise product of the object and the depth mask. Only objects with an average reachable distance smaller than 1.5 meter is stored. The confidence score of which our agent can act on an object is calculated using the sum of their mask. This score gives our egocentric planner a way to prioritize which object to interact with.

F.2 Semantic Spatial Graph

The spatial graph plays the role of an agent's memory during exploration and bridges the gap between grounded objects and the planner's state representation requirements. The exploration employs an egocentric agent who only has direct access to its local space where action can be executed within a distance threshold of 1.5 meters. The key for each node is encoded as location, with visual observations being the values. The edges represent the agent's actions, comprising four specific actions: MOVEAHEAD, MOVEBACK, TURNLEFT, and TURNRIGHT, with movement actions having a step size of 0.25 meter and turning actions featuring a turn angle of 90°. This spatial graph is updated and expanded over time through policies generated by our egocentric planner, with each node storing object classes, segmentation masks, and depth information produced via the vision module. Fig. 2 shows an example of a semantic spatial graph.

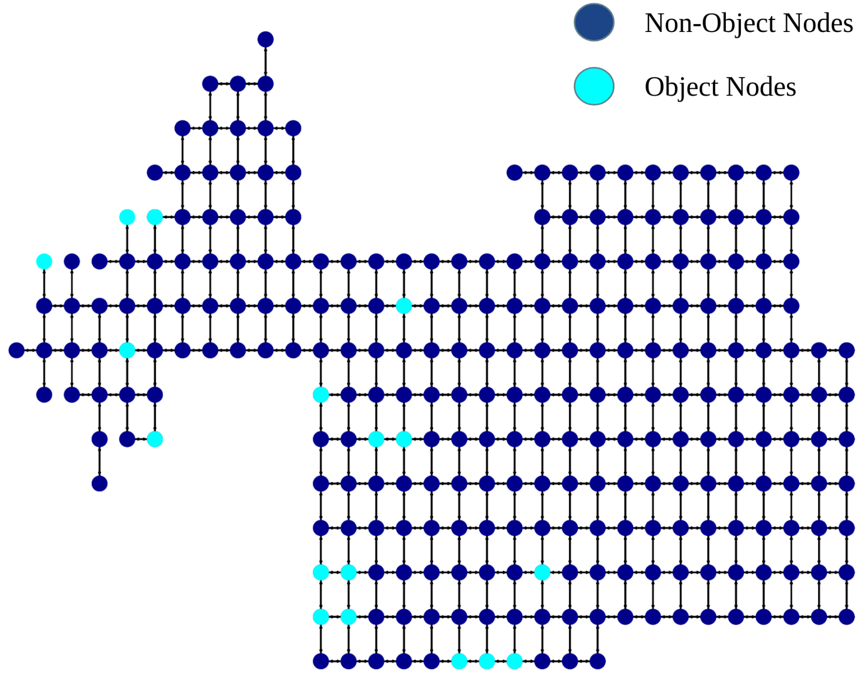

Figure 2: Example of a Semantic Spatial Graph after partial Exploration. The agent is at one of the nodes.

The spatial graph undergoes continuous updates and expansions as a result of the policies generated by our egocentric planner. It involves the extraction and storage of objects within the graph, subject to a confidence score threshold of 0.8. Objects that meet or exceed this confidence score are deemed reliable and considered for storage. Each node within the graph maintains a record of various attributes associated with the objects, including object classes, segmentation masks, and depth information. The object classes provide information about the category or type of each object, allowing for efficient categorization and analysis. The segmentation masks of the extracted objects are stored as a list of pixel coordinates. These masks outline the boundaries of each object within the captured visual data. By storing the masks in this format, we can later retrieve and utilize the spatial information of each object during subsequent analyses or interactions. Moreover, the distance to each object is calculated by leveraging the predicted distance of the top 100 pixels with reference to the depth mask generated by the U-net model. The U-net model is responsible for producing a depth mask that estimates the distances of various points within the scene. By employing the predicted distance values of the top 100 pixels associated with each object, we obtain an approximate measure of its distance from the egocentric viewpoint. Actionable objects are treated as affordances at each node, allowing the agent to perform desired actions. Nodes with objects of interest are ranked based on the confidence scores outlined earlier output by the MaskRCNN.

F.3 Egocentric Algorithm for ALFRED

The egocentric planning algorithm for ALFRED uses location as the only anchor types $\mathcal{T}_a$, and the move action as the only exploration action $\mathcal{X}$. Consequently, exploration plans employ move actions to visit unexplored locations, revealing new objects visible from such locations. As the agent explores, the mental model creates new objects and adds them to the initial state, facilitating the incremental construction of a semantic spatial graph, which maps the environment and object locations. An ALFRED task, $T_{\mathcal{E}} = \langle I_{\mathcal{E}}, G_{\mathcal{E}} \rangle$, is characterized by a goal that corresponds to one of the seven tasks detailed in section 2, and involves specific predicates for the task-related objects. The goal is expressed as an existential formula over potential objects, as it refers to objects not visible from the initial state (Listings 2).

We also implemented some ALFRED-specific optimizations. For example, we mitigate the risk of executing irreversible actions by only attempting them after devising a plan projected to achieve the goal. This strategy implies that some actions are unavailable during the exploration phase. For each time step, our egocentric planner will produce a goal or exploration action depending on whether the agent has sufficient information to achieve its goal. The replanning loop repeats when an action fails. Priority are given to unknown location based on objects of interest for a particular task and number of total objects found in an area. They are ranked base on number of such objects. Information are gathered for each action taken which is stored in the semantic map. In order to reduce the amount of overall exploration required at the online planning stage, we allow our agent to first conduct 500 random exploration movements. The semantic graph after exploration is used as the initial state for our egocentric planning setup. To enhance efficiency, we tailor the planning domain to the specific task, disregarding actions unlikely to contribute to reasonable plans to achieve the goal. This optimization can be automatically implemented via reachability analysis, a technique commonly used in classical planning to prevent grounding actions irrelevant to achieving the goal from the initial state (Hof01).

F.4 PDDL Example

Below we provide examples assuming that the natural language description of the task is "heat potato in the microwave and put the potato on the table." We start by generating the planning setup for the egocentric planner, given the task identified as "pick-heat-then-place-in-recep." The objects involved in this manipulation are classified as "MicrowaveType", "PotatoType" and "TableType". The egocentric planner will eventually produce a sequence of actions to accomplish the desired task. In this case, the steps might include picking up the potato, heating it, and then placing it on the table.

Listings 1 and 2 show the PDDL domain and problem files for the example task. In a PDDL domain for classical planning, both predicates and actions arguments are expressed as a list of arguments, each one a variable with an optional type. For instance, `(conn ?x1 - location ?x2 - location ?x3 - movement)` express a predicate `conn` with three arguments: two locations and a kind movement: ahead, rotate-left or rotate-right. In a PDDL problem for classical planning, object constants are associated with an optional type, and the initial state is a list of predicates grounded on existing constants. For instance, `l1 l2 l3 - location` means three constants of type `location`. The initial state could be `(and (at l1) (conn l1 l2 ahead))`. An action `(MoveAgent ?from ?to - location ?dir - movement)` has as a precondition that the agent is at the location `?from` and both locations are contiguous in the direction `?dir`. Executing the action sets to false that the agent is at the location `?from`, and sets to true that the agent is at the location `?to`. Classical plans in PDDL are sequences of actions grounded on constants. For instance, a first possible action might be `(MoveAgent l1 l2 ahead)`, leading to the state `(and (at l2) (conn l1 l2 ahead))`. For an introduction to PDDL see (HLMM19).

Listing 1: ALFRED PDDL domain file

```
1  (define (domain alfred_task)
2      (:requirements :equality :typing)
3      (:types
4          agent - object
5          location - object
6          receptacle - object
7          obj - object
8          itemtype - object
9          movement - object
10     )
11
12     (:predicates
13         (conn ?x1 - location ?x2 - location ?x3 - movement)
14         (move ?x1 - movement)
15         (atLocation ?x1 - agent ?x2 - location)
16         (receptacleAtLocation ?x1 - receptacle ?x2 - location)
17         (objectAtLocation ?x1 - obj ?x2 - location)
18         (canContain ?x1 - itemtype ?x2 - itemtype)
19         (inReceptacle ?x1 - obj ?x2 - receptacle)
20         (recepInReceptacle ?x1 - receptacle ?x2 - receptacle)
```

```
21        (isInReceptacle ?x1 - obj)
22        (openable ?x1 - itemtype)
23        (canHeat ?x1 - itemtype ?x2 - itemtype)
24        (isHeated ?x1 - obj)
25        (canCool ?x1 - itemtype ?x2 - itemtype)
26        (isCooled ?x1 - obj)
27        (canClean ?x1 - itemtype ?x2 - itemtype)
28        (isCleaned ?x1 - obj)
29        (isMovableReceptacle ?x1 - receptacle)
30        (receptacleType ?x1 - receptacle ?x2 - itemtype)
31        (objectType ?x1 - obj ?x2 - itemtype)
32        (holds ?x1 - obj)
33        (holdsReceptacle ?x1 - receptacle)
34        (holdsAny)
35        (unknown ?x1 - location)
36        (explore)
37        (canToggle ?x1 - obj)
38        (isToggled ?x1 - obj)
39        (canSlice ?x1 - itemtype ?x2 - itemtype)
40        (isSliced ?x1 - obj)
41        (emptyObj ?x1 - obj)
42        (emptyReceptacle ?x1 - receptacle)
43    )
44
45    (:action explore_MoveAgent
46        :parameters (?agent0 - agent ?from0 - location
47            ?loc0 - location ?mov0 - movement)
48        :precondition (and (atLocation ?agent0 ?from0)
49            (move ?mov0) (conn ?from0 ?loc0 ?mov0)
50            (unknown ?loc0) (not (explore)))
51        :effect (and
52            (not (atLocation ?agent0 ?from0))
53            (atLocation ?agent0 ?loc0)
54            (not (unknown ?loc0))
55            (explore))
56    )
57
58    (:action MoveAgent
59        :parameters (?agent0 - agent ?from0 - location
60            ?loc0 - location ?mov0 - movement)
61        :precondition (and (atLocation ?agent0 ?from0)
62            (move ?mov0) (conn ?from0 ?loc0 ?mov0))
63        :effect (and
64            (not (atLocation ?agent0 ?from0))
65            (atLocation ?agent0 ?loc0))
66    )
67
68    (:action pickupObject
69        :parameters (?agent0 - agent ?loc0 - location ?obj0 - obj)
70        :precondition (and (atLocation ?agent0 ?loc0)
71            (objectAtLocation ?obj0 ?loc0)
72            (not (isInReceptacle ?obj0)) (not (holdsAny)))
73        :effect (and
74            (not (objectAtLocation ?obj0 ?loc0))
75            (holds ?obj0)
76            (holdsAny))
77    )
78
79    (:action pickupObjectFrom
80        :parameters (?agent0 - agent ?loc0 - location
81            ?obj0 - obj ?recep0 - receptacle)
82        :precondition (and (atLocation ?agent0 ?loc0)
83            (objectAtLocation ?obj0 ?loc0) (isInReceptacle ?obj0)
84            (inReceptacle ?obj0 ?recep0) (not (holdsAny)))
85        :effect (and
```

```
            (not (objectAtLocation ?obj0 ?loc0))
            (holds ?obj0)
            (holdsAny)
            (not (isInReceptacle ?obj0))
            (not (inReceptacle ?obj0 ?recep0)))
    )

    (:action putonReceptacle
        :parameters (?agent0 - agent ?loc0 - location ?obj0 - obj
            ?otype0 - itemtype ?recep0 - receptacle
            ?rtype0 - itemtype)
        :precondition (and
            (atLocation ?agent0 ?loc0)
            (receptacleAtLocation ?recep0 ?loc0)
            (canContain ?rtype0 ?otype0) (objectType ?obj0 ?otype0)
            (receptacleType ?recep0 ?rtype0)
            (holds ?obj0) (holdsAny) (not (openable ?rtype0))
            (not (inReceptacle ?obj0 ?recep0)))
        :effect (and
            (not (holdsAny))
            (not (holds ?obj0))
            (isInReceptacle ?obj0)
            (inReceptacle ?obj0 ?recep0)
            (objectAtLocation ?obj0 ?loc0))
    )

    (:action putinReceptacle
        :parameters (?agent0 - agent ?loc0 - location ?obj0 - obj
            ?otype0 - itemtype ?recep0 - receptacle
            ?rtype0 - itemtype)
        :precondition (and
            (atLocation ?agent0 ?loc0)
            (receptacleAtLocation ?recep0 ?loc0)
            (canContain ?rtype0 ?otype0) (objectType ?obj0 ?otype0)
            (receptacleType ?recep0 ?rtype0) (holds ?obj0)
            (holdsAny) (openable ?rtype0)
            (not (inReceptacle ?obj0 ?recep0)))
        :effect (and
            (not (holdsAny))
            (not (holds ?obj0))
            (isInReceptacle ?obj0)
            (inReceptacle ?obj0 ?recep0)
            (objectAtLocation ?obj0 ?loc0))
    )

    (:action toggleOn
        :parameters (?agent0 - agent ?loc0 - location ?obj0 - obj)
        :precondition (and (atLocation ?agent0 ?loc0)
            (objectAtLocation ?obj0 ?loc0) (not (isToggled ?obj0)))
        :effect (and
            (isToggled ?obj0))
    )

    (:action heatObject
        :parameters (?agent0 - agent ?loc0 - location ?obj0 - obj
            ?otype0 - itemtype ?recep0 - receptacle
            ?rtype0 - itemtype)
        :precondition (and
            (atLocation ?agent0 ?loc0)
            (receptacleAtLocation ?recep0 ?loc0)
            (canHeat ?rtype0 ?otype0)
            (objectType ?obj0 ?otype0)
            (receptacleType ?recep0 ?rtype0)
            (holds ?obj0) (holdsAny))
        :effect (and
```

```
151              (isHeated ?obj0))
152     )
153 )
```

Next, Listings 2 shows the generated the planning problem based on the initial observation. This involves specifying the initial state of the environment and the goal that we want to achieve through the planning process.

Listing 2: ALFRED PDDL problem file with task achievement goal

```
1  (define (problem problem0)
2      (:domain alfred_task)
3
4      (:objects
5          agent0 - agent
6          AlarmClockType AppleType ArmChairType BackgroundType
7          BaseballBatType BasketBallType BathtubBasinType BathtubType
8          BedType BlindsType BookType BootsType BowlType BoxType
               BreadType
9          ButterKnifeType CDType CabinetType CandleType CartType
10         CellPhoneType ... - itemtype
11         f0_0_0f f0_0_1f f0_0_2f f0_0_3f f0_1_0f - location
12         MoveAhead RotateLeft RotateRight - movement
13         emptyO - obj
14         emptyR - receptacle
15     )
16
17     (:init
18         (emptyObj emptyO)
19         (emptyReceptacle emptyR)
20         (canContain BedType CellPhoneType)
21         (canContain CounterTopType PotatoType)
22         (canContain CoffeeTableType StatueType)
23         (canContain DiningTableType PanType)
24         ....
25         (openable MicrowaveType)
26         (atLocation agent0 f0_0_0f)
27         (move RotateRight)
28         (move RotateLeft)
29         (move MoveAhead)
30         (canHeat MicrowaveType PlateType)
31         (canHeat MicrowaveType PotatoType)
32         (canHeat MicrowaveType MugType)
33         (canHeat MicrowaveType CupType)
34         (canHeat MicrowaveType BreadType)
35         (canHeat MicrowaveType TomatoType)
36         (canHeat MicrowaveType AppleType)
37         (canHeat MicrowaveType EggType)
38         (conn f0_0_1f f0_0_0f RotateLeft)
39         (conn f0_0_3f f0_0_2f RotateLeft)
40         (conn f0_0_0f f0_0_1f RotateRight)
41         (conn f0_0_1f f0_0_2f RotateRight)
42         (conn f0_0_2f f0_0_3f RotateRight)
43         (conn f0_0_0f f0_1_0f MoveAhead)
44         (conn f0_0_2f f0_0_1f RotateLeft)
45         (conn f0_0_3f f0_0_0f RotateRight)
46         (conn f0_0_0f f0_0_3f RotateLeft)
47         (unknown f0_1_0f)
48     )
49
50     (:goal
51         (exists
52             (?goalObj - obj ?goalReceptacle - receptacle)
53             (and (inReceptacle ?goalObj ?goalReceptacle)
54                 (objectType ?goalObj PotatoType)
```

```
55                    (receptacleType ?goalReceptacle TableType)
56                    (isHeated ?goalObj)))
57      )
58 )
```

As not plan was found for the problem in Listing 2, the goal is changed to exploration in listing 3.

Listing 3: ALFRED PDDL problem file with exploration goal

```
1      (:goal
2          (and (explore ) (not (holdsAny )))
3      )
```