Learning Rational Subgoals from Demonstrations and Instructions

Zhezheng Luo*[1], Jiayuan Mao*[1], Jiajun Wu[2],
Tomás Lozano-Pérez[1], Joshua B. Tenenbaum[1], Leslie Pack Kaelbling[1]
[1] Massachusetts Institute of Technology [2] Stanford University

Abstract

We present a framework for learning useful subgoals that support efficient long-term planning to achieve novel goals. At the core of our framework is a collection of rational subgoals (RSGs), which are essentially binary classifiers over the environmental states. RSGs can be learned from weakly-annotated data, in the form of *unsegmented* demonstration trajectories, paired with abstract task descriptions, which are composed of terms initially unknown to the agent (e.g., *collect-wood* then *craft-boat* then *go-across-river*). Our framework also discovers dependencies between RSGs, e.g., the task *collect-wood* is a helpful subgoal for the task *craft-boat*. Given a goal description, the learned subgoals and the derived dependencies facilitate off-the-shelf planning algorithms, such as A^* and RRT, by setting helpful subgoals as waypoints to the planner, which significantly improves performance-time efficiency. Project page: https://rsg.csail.mit.edu

Introduction

Being able to decompose complex tasks into subgoals is critical for efficient long-term planning. Consider the example in Fig. 1: planning to craft a boat from scratch is hard, as it requires a long-term plan going from collecting materials to crafting boats, but it can be made easier if we know that *having an axe* and *having wood* are useful sub-goals. Planning hierarchically with these subgoals can substantially reduce the search required. It is also helpful to understand the temporal dependencies between these subgoals, such as *having wood* being a useful subgoal to achieve *prior to crafting boat* makes long-term planning much more efficient.

In this work, we propose *Rational Subgoals* (RSGs), a framework for learning useful subgoals and their temporal dependencies from demonstrations. Our system learns with very weak supervision, in the form of a small number of *unsegmented* demonstrations of complex behaviors paired with abstract task descriptions. The descriptions are composed of terms that are initially unknown to the agent, much as an adult might narrate the high-level steps when demonstrating a cooking recipe to a child. These action terms indicate important *subgoals* in the action sequence, and our agent learns to detect when these subgoals are true in the world, infer their temporal dependencies, and leverage them to plan efficiently.

Illustrated in Fig. 1, our model learns from a dataset of paired but unaligned low-level state-action sequences and the corresponding abstract task description (*collect-wood* then *craft-boat* then *go-across-river*). For each action term o (e.g., *collect-wood*), our model learns a goal condition G_o, which maps any state to a binary random variable, indicating whether the state satisfies the goal condition. Given the training data, we decompose the observed trajectory into fragments, each of which corresponds to a "rational" sequence of actions for achieving a subgoal in the description.

While this model-based approach enables great generality in generating behaviors, it suffers from the slow online computation. To speed up online planning, we compute a dependency matrix whose entries encode which subgoals might be helpful to achieve before accomplishing another subgoal (e.g., *having wood* is a helpful subgoal for the task *crafting boat*, and thus the entry (*having wood*, *crafting boat*) will have a higher weight). During test time, given a final goal (e.g., *craft boat*) and the initial state, a hierarchical search algorithm is applied at both the subgoal level and the lower, environmental-action level.

The explicit learning of subgoals and their dependency structures brings two important advantages. First, the subgoal dependency allows us to explicitly set helpful subgoals as waypoints for planners. This significantly improves their runtime efficiency. Second, compared to alternative subgoal parameterizations such as reward functions, subgoals in the form of a state classifier allows us to use simple and efficient planners. For example, in continuous spaces, we can use Rapidly-exploring Random Trees (RRT) to search for plans in the robot configuration space. These planers do not require training and generalize immediately to novel environments.

We evaluate RSGs in Crafting World (Chen, Gupta, and Marino 2021), an image-based grid-world domain with a rich set of object crafting tasks, and Playroom (Konidaris, Kaelbling, and Lozano-Perez 2018), a 2D continuous domain with geometric constraints. Our evaluation shows that our model clearly outperforms baselines on planning tasks where the agent needs to generate trajectories to accomplish a given task. Another important application of RSGs is to create a language interface for human-robot communication, which includes robots interpreting human actions and humans in-

*These authors contributed equally.

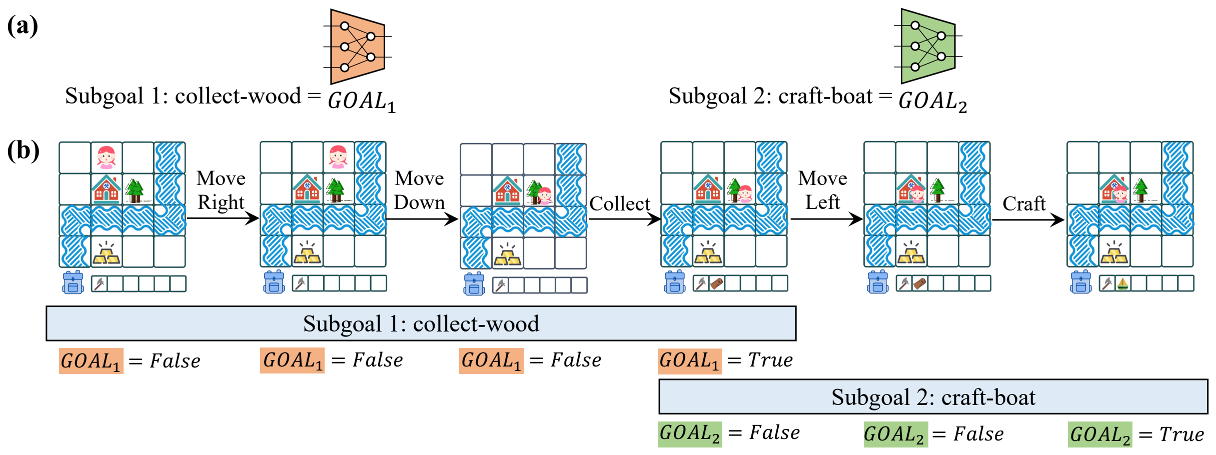

Figure 1: Interpreting a demonstration and its description in terms of RSGs: (a) Each RSG is represented as a subgoal G_o. (b) The system infers a transition to the next subgoal if the G condition is satisfied. Such transition rules can be used to interpret demonstrations and to plan for tasks that require multiple steps to achieve.

structing robots by specifying a sequence of subgoals. Our model enables compositional generalization through flexible re-composition of learned subgoals, which allows the robot to interpret and execute novel instructions.

Rational Subgoal Learning and Planning

We focus on learning rational subgoals from demonstration data and leveraging them for planning. Formally, our training data is a collection of paired *unsegmented* demonstrations (i.e., state and action sequences) and abstract descriptions (e.g., *collect-wood* <u>*then*</u> *craft-boat*) composed of action terms (*collect-wood*, etc.) and connectives (<u>*then*</u>, <u>*or*</u>). Our ultimate goal is to recover the *grounding* (i.e., the corresponding subgoal specified by the action term) for each individual action term. These subgoals will be leveraged by planning algorithms to solve long-horizon planning problems.

We begin this section with basic definitions of the rational subgoal representations and the language $\mathcal{TL}$ for abstract descriptions. Second, we outline the planning algorithm we use to refine high-level instructions in $\mathcal{TL}$ into environmental actions that agents can execute, *given* the RSGs. Although any search algorithms or Markov Decision Process (MDP) solvers are in principle applicable for our planning task, in this paper, we have focused on a simple extension to the A* algorithm. Next, we present the algorithm we use to *learn* RSGs from data. Since we are working with unsegmented trajectories, the learning algorithm has two steps. It first computes a rationality score for individual actions in the trajectory based on the optimal plan derived from the A* algorithm. Then, it uses a dynamic programming algorithm to find the best segmentation of the trajectory and updates the parameters. Finally, we describe a dependency discovery algorithm for RSGs and apply it to solve planning tasks given only a single goal action term (e.g., *collect-gold*), in contrast to the earlier case where there are detailed step-by-step instructions.

We call our representation *rational* subgoals because our learning algorithm is based on a *rationality* objective with

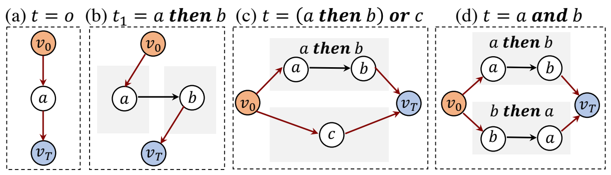

Figure 2: Illustrative example of how finite state machines (FSM) are constructed from task descriptions. The super-starting node v_0 and the super-terminal node v_T are highlighted.

respect to demonstration trajectories, and our planning algorithm chooses *rational* subgoals to accelerate the search.

Formally, a rational subgoal (RSG) is a classifier that maps an environmental state s to a Boolean value, indicating whether the goal condition is satisfied at s. Each RSG has an atomic name o (e.g., *collect-wood*), and the corresponding goal classifier is denoted by G_o. Depending on the representation of states, G_o can take various forms of neural networks, such as convolutional neural networks (CNNs) for image-based state representations.

In both learning and planning, we will be using an abstract language to describe tasks, such as *collect-wood* <u>*then*</u> *craft-boat*. These descriptions are written in a formal task language $\mathcal{TL}$. Syntactically, all atomic subgoals are in $\mathcal{TL}$; and for all $t_1, t_2 \in \mathcal{TL}$, $(t_1 \ \underline{then}\ t_2)$, $(t_1 \ \underline{or}\ t_2)$, and $(t_1 \ \underline{and}\ t_2)$ are in $\mathcal{TL}$. Semantically, a state sequence $\bar{s}$ satisfies a task description t, written $\bar{s} \models t$ when:

- If t is a *RSG* o, then the first state does not satisfy G_o, and the last state satisfies G_o. Note that this implies that the sequence $\bar{s}$ must have at least 2 states.
- If $t = (t_1 \ \underline{then}\ t_2)$ then $\exists 0 < j < n$ such that $(s_1, \dots, s_j) \models t_1$ and $(s_j, \dots, s_n) \models t_2$: task t_1 should be accomplished before t_2.
- If $t = (t_1 \ \underline{or}\ t_2)$ then $\bar{s} \models t_1$ or $\bar{s} \models t_2$: the agent should either complete t_1 or t_2.

- If $t = (t_1$ *and* $t_2)$ then $\bar{s} \models (t_1$ *then* $t_2)$ or $\bar{s} \models (t_2$ *then* $t_1)$: the agent should complete both t_1 and t_2, but in any order (t_1 first or t_2 first)[*].

Note that the relation $\bar{s} \models t$ only specifies whether $\bar{s}$ completes t but not how optimal $\bar{s}$ is. Later on, when we define the planning problem, we will introduce the trajectory cost.

Each task description $t \in \mathcal{TL}$ can be represented with a non-deterministic finite state machine (FSM), representing the sequential and branching structures. Each FSM_t is a tuple (V_t, E_t, VI_t, VG_t) which are subgoal nodes, edges, set of possible starting nodes and set of terminal nodes. Each node corresponds to an action term in the description, and each edge corresponds to a possible transition of changing subgoals. Fig. 2 illustrates the constructions for syntax in $\mathcal{TL}$, and we provide the follow algorithm for the construction.

- **Single subgoal:** A single subgoal s is corresponding FSM with a single node i.e. $VI_t = VG_t = V_t = \{s\}$, and $E_t = \emptyset$.
- t_1 *then* t_2: We merge FSM_{t_1} and FSM_{t_2} by merging their subgoal nodes, edges and using VI_{t_1} as the new starting node set and VG_{t_2} as the new terminal node set. Then, we add all edges from VG_{t_1} to VI_{t_2}. Formally,

$$\text{FSM}_{t_1 \text{ \underline{then} } t_2} = (V_{t_1} \cup V_{t_2}, E_{t_1} \cup E_{t_2} \cup (VG_{t_1} \times VI_{t_2}), VI_{t_1}, VG_{t_2}),$$

  where $\times$ indicates the Cartesian product, meaning that each terminal node of FSM_{t_1} can transit to any starting node of FSM_{t_2}.
- t_1 *or* $\cdots$ *or* t_n: Simply merge n FSMs without adding any new edges. Formally,

$$\text{FSM}_{t_1 \text{ or } \cdots \text{ or } t_n} = (\bigcup_i V_{t_i}, \bigcup_i E_{t_i}, \bigcup_i VI_{t_i}, \bigcup_i VG_{t_i})$$

- t_1 *and* $\cdots$ *and* t_n: Build $2^{n-1}n$ sub-FSMs over n layers: the i-th layer contains $n \cdot \binom{n-1}{i-1}$ sub-FSMs each labeled by (s, D) where s is the current subgoal to complete (so this sub-FSM is a copy of FSM_s), and D is the set of subgoals that have been previously completed. Then for a sub-FSM (s_1, D_1) and a sub-FSM (s_2, D_2) in the next layer, if $D_2 = D_1 \cup \{s_1\}$, we add all edges from terminal nodes of the first sub-FSM to starting nodes of the second sub-FSM. After building layers of sub-FSMs and connecting them, we set the starting nodes to be the union of starting nodes in the first layer and terminal nodes to be the union of terminal nodes in the last layer.

Note that our framework requires the starting and terminal nodes to be unique, but the construction above may output a FSM with multiple starting/terminal nodes, so we introduce the virual super starting node v_0 and terminal node v_T to unify them.

[*]The operator *and* can be generalized be n-ary. In this case, accomplishing them in any order is considered accomplishing the composed task. For example, the task *mine-wood* *and* *mine-gold* *and* *mine-coal* allows the agent to accomplish all three subgoals in any order. Note that this is different from the specification with parenthesis: *(mine-wood and mine-gold) and mine-coal*.

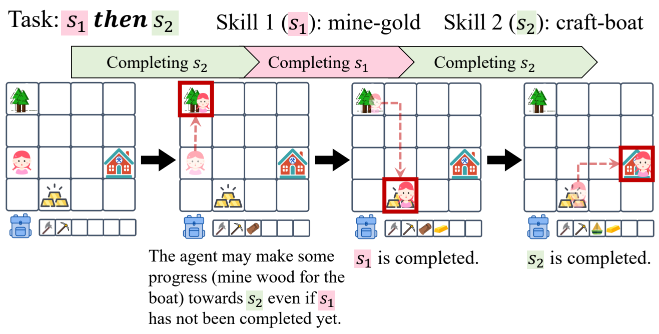
Task: s_1 ***then*** s_2
Skill 1 (s_1): mine-gold
Skill 2 (s_2): craft-boat
Completing s_2
Completing s_1
Completing s_2
The agent may make some progress (mine wood for the boat) towards s_2 even if s_1 has not been completed yet.
s_1 is completed.
s_2 is completed.

Figure 3: An example of optimal interleaving subgoals: s_1 is "mine gold", and s_2 is "craft boat". It is valid that the agent first goes to collect wood (for accompolishing s_2), and then mine gold (for accompolishing s_1), and finally crafts boat. In this case, the action sequences for completing s_1 and s_2 are interleaved. However, they can are be recognized as s_2 *then* s_2 because s_1 is accomplished before s_2.

Remark. In this paper, the language $\mathcal{TL}$ used for describing tasks covers LTL$_f$, a finite fragment of LTL that does not contain the *always* quantifier, so our fragment does not model task specifications that contain infinite loops. Finite LTL formulae can be converted to a finite automaton (De Giacomo and Vardi 2013), represented using the FSM.

Execution steps for different subgoals can interleave. RSGs does not simply run optimal policy for each individual subgoal sequentially. Rather, the semantic of s_1 *then* s_2 is: s_1 should be completed before s_2. It does not restrict the agent from making progress towards the subgoal before the subgoal is completed. In some case, such interleaving is necessary to obtain the globally optimal trajectory.

Consider the example shown in Figure 3, where s_1 is "mine-gold", and s_2 is "craft-boat". It is valid that the agent first goes to collect wood (for accompolishing s_2), and then mine gold (for accompolishing s_1), and finally crafts boat. In this case, the action sequences for completing s_1 and s_2 are interleaved. However, they can are be recognized as s_1 *then* s_2 because s_1 is accomplished before s_2.

Planning with RSGs

We first consider the problem of planning an action sequence that satisfies a given task description t written in $\mathcal{TL}$. We assume that the external world is well modeled as a deterministic, fully observable decision process with a known state space, an action space, a transition function, and a cost function $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C} \rangle$ and that we have a set of goal classifiers G_o parameterized by θ. Given a task t, we construct an FSM representation and then compose it with the environment process to obtain an FSM-augmented process $\langle \mathcal{S}_t, \mathcal{A}_t, \mathcal{T}_t, \mathcal{C}_t \rangle$. Concretely, $\mathcal{S}_t = \mathcal{S} \times V_t$, where V_t is the set of nodes of FSM constructed from task t. We then denote each task-augmented state as (s, v), where s is the environment state, and v indicates the current subgoal. The actions $\mathcal{A}_t = \mathcal{A} \cup \text{FSM}_t$, where each action either corresponds to a primitive action $a \in \mathcal{A}$ or a transition in FSM_t. An FSM tran-

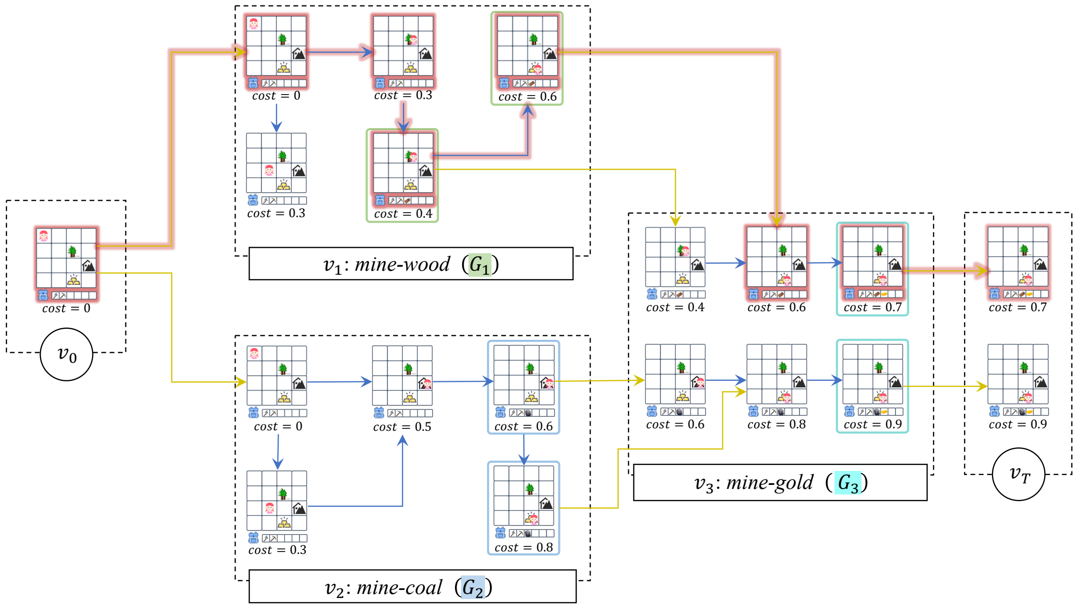

Figure 4: A running example of the FSM-A^* algorithm for the task "*(mine wood* *or* *mine coal)* *then* *mine gold*." For simplicity, we only show a subset of states visited on each FSM node. The blue arrows indicate transitions by primitive actions (in this example, each primitive action takes a cost of 0.1). The yellow arrows are transitions on the FSM, which can only be performed when $G_v(\cdot)$ and $G_{v'}(\cdot)$ evaluates to False (in practice, the reward is computed as $-(\log G_v(\cdot) + \log(1 - G_{v'}(\cdot)))$). At the super-terminal node v_T, the state with minimum cost will be selected and we will back-trace the entire state-action sequence.

sition action indicates that the agent has achieved the current subgoal and will proceed to the next subgoal. We further define $\mathcal{T}_t((s,v),a) = (\mathcal{T}(s,a),v)$ if a is a primitive action in $\mathcal{A}$, while $\mathcal{T}_t((s,v),a) = (s,v')$ if $a = (v,v') \in \text{FSM}_t$ is an edge in the FSM. The former are environmental actions. They only change the environmental state s but do not change the current subgoal v. The latter, namely FSM transitions, do not change the environmental state, but mark the current subgoal as completed and switch to the next one. Similarly, for the cost function,

$$\mathcal{C}'((s,v),a) = \begin{cases} \mathcal{C}(s,a) & \text{if } a \in \mathcal{A}, \\ -\lambda(\log G_v(s;\theta) + & \text{if } a = (v,v') \in \text{FSM}_t \\ \quad \log(1 - G_{v'}(s;\theta))) & \end{cases}$$

where λ is a hyperparameter. The key intuition behind the construction of $\mathcal{C}_t$ is that the cumulative cost from v_0 to v_T is the summation of all primitive action costs added to the log probability of the validity of subgoal transitions. At each subgoal transition, the state s should satisfy the goal condition of the current RSGs but should not satisfy the goal condition of the next RSGs—which enforces the sequential constraints specified in the task. In principle, when G_v are Boolean-output classifiers, the cost is 0 for a valid transition and ∞ for an invalid transition. In practice, we approximate the "soft" version of classifiers with neural networks: the outputs are in $[0,1]$, indicating how likely those conditions are to be satisfied.

Importantly, our formulation of the RSG planning problem is different from planning for each individual action term and stitching the sub-plans sequentially. Concretely, we are finding a "globally" optimal plan instead of achieving individual subgoals in a locally optimal way. Thus, we allow complex behaviors such as making progress for a later subgoal to reduce the total cost. We include detailed examples in the supplementary material.

At the input-output level, our planner receives the a task description t represented as an FSM, an environmental transition model $\mathcal{T}$, and a cost function $\mathcal{C}$, together with a set of goal classifiers $\{G_o\}$ parameterized by θ. It generates a sequence of actions $\bar{a}$ that is a path from (s_0, v_0) to (s_T, v_T) and minimizes the cumulative action costs defined by $\mathcal{C}_t$. Here, s_0 is the initial environmental state, v_0 is the initial state of FSM_t, s_T is the last state of the trajectory, and v_T is the terminal state of FSM_t.

We make plans using slightly modified versions of A^* search, with a learned domain-dependent heuristic for previously seen tasks and a uniform heuristic for unseen tasks. This algorithm can be viewed as doing a forward search to construct a trajectory from a given state to a state that satisfies the goal condition. Our extension to the algorithms handles the hierarchical task structure of the FSM.

Our modified A^* search maintains a priority queue of nodes to be expanded. At each step, instead of always popping the task-augmented state (s,v) with the optimal evaluation, we first sample a subgoal v uniformly in the FSM, and then choose the priority-queue node with the smallest evaluation value among all states $(\cdot, v)$. This balances the time allocated to finding a successful trajectory for each subgoals in the task description.

Our hierarchical search algorithm also extends to continuous domains by integrating Rapidly-Exploring Random Trees (RRT) (LaValle et al. 1998). We include the implementation details in the supplementary material. Any state-action sequence produced by planning in the augmented model is legal according to the environment transition model and is guaranteed to satisfy the task specification t.

Example. Fig. 4 shows a running example of our FSM-A^* planning given the task "*mine wood* or *mine coal* then *mine gold*" from the state s_0 (shown as the left-most state in the figure).

1. At the beginning, (s_0, v_0) is expanded to the node v_1:*mine wood* and v_2:*mine coal* with FSM transition actions at no cost.
2. We expand the search tree node on v_1 and v_2 and compute the cost for reaching each states on v_1 and v_2.
3. For states that satisfy the goal conditions for v_1 and v_2 (i.e., G_1 and G_2, respectively, and circled by green and blue boxes) and the initial condition for v_3 (i.e., $1 - G_3$), we make a transition to v_3 at no cost (the states that do not satisfy the conditions can also be expanded to v_3 but with a large cost.
4. Then search can be done in a similar way at v_3 and the states at v_3 that satisfy G_3 can reach v_T.
5. For all states at v_T, we back-trace the state sequence with the minimum cost.

Learning RSGs from Unsegmented Trajectories and Descriptions

We learn RSGs from weakly-annotated demonstrations, in the form of *unsegmented* trajectories and paired task descriptions. The training dataset $\mathcal{D}$ contains tuples $(\bar{s}, \bar{a}, t)$ where $\bar{s}$ is a sequence of environmental states, $\bar{a}$ is a sequence of actions, and $t \in \mathcal{TL}$ is a task description.

Our goal is to recover the grounding of subgoal terms from these demonstrations. At a high level, our learning objective is to find a set of parameters for the goal classifiers G_o that *rationally* explain the demonstration data: the actions taken by the demonstrator should be "close" in some sense to the optimal actions that would be taken to achieve the goal. Let θ denote the collection of parameters in $\{G_o\}$. Thus, our training objective takes the following form:

$$\theta^* = \arg\max_\theta \frac{1}{|\mathcal{D}|} \sum_{(\bar{s},\bar{a},t)\in\mathcal{D}} \textit{score}\left(\bar{s}, \bar{a}, t; \theta\right) \ . \qquad (1)$$

The scoring function *score* combines the *rationality* of the observed trajectory with an additional term that emphasizes the appropriateness of FSM transitions given t:

$$\textit{score}(\bar{s}, \bar{a}, t; \theta) := \max_{\bar{v}} \Big\{ \log \prod_i \mathrm{Rat}\left(s_i, v_i, a_i, t; \theta\right) + \sum_{\substack{(v_i, v_{i+1})\in \\ \text{FSM transitions}}} \left\{ \log G_{v_i}(s_i; \theta) + \log\left(1 - G_{v_{i+1}}(s_i; \theta)\right)\right\} \Big\} \qquad (2)$$

The rationality score measures the likelihood that the action $a \in \mathcal{A}_t$ in state (s, v) would have been chosen by a nearly-optimal agent, who is executing a policy that assigns a probability to an action based on the optimal cost-to-go for task t in the FSM-augmented model after taking it:

$$\mathrm{Rat}\left(s, v, a, t; \theta\right) := \frac{\exp\left(-\alpha \cdot J_t(s, v, a; \theta)\right)}{\int_{x\in\mathcal{A}'} \exp\left(-\alpha \cdot J_t(s, v, x; \theta)\right)}, \qquad (3)$$

where α is a hyperparameter called inverse rationality. The integral is a finite sum for discrete actions and can be approximated using Monte Carlo sampling for continuous actions. If α is small, the assumption is that the demonstrations may be highly noisy; if large, then they are near optimal.

The cost-to-go (analogous to a value function) is defined recursively as

$$J_t(s, v, a; \theta) = \mathcal{C}_t\left((s, v), a\right) + \max_{a'\in\mathcal{A}_t} J_t\left(\mathcal{T}'\left((s, v), a\right), a; \theta\right). \qquad (4)$$

It need not be computed for the whole state space; rather, it can be computed using the planner on a tree of relevant states, reachable from (s_0, v_0).

Figure 5 and Algorithm 1 summarize the learning process of RSGs. First, we perform a A^* search (or RRT for continuous domains) from the trajectory. Then, we backtrack in the search tree/RRT to compute the shortest distance from each node to the terminal state, J_t, so that $\mathrm{Rat}(s_i, v_i, a_i, t; \theta)$ can be evaluated along the trajectory $\bar{s}, \bar{a}$.

At learning time, we can observe the environmental state and action sequence, but we cannot observe the FSM states or transitions. To efficiently find the optimal FSM states and transitions, given an environment state and action sequence as well as goal classifiers parameterized by the current θ, we use a dynamic programming method. Specifically, we will first label the FSM nodes from 0 to T by sorting them topologically. Next, we can use a two-dimensional dynamic programming with the transition equations based on Rat and G_v can find $\bar{v}$ that maximizes *score*. Concretely, let $f[i, j]$ denote the maximum score by aligning the trajectory $s_i, a_i, s_{i+1}, \cdots$ with the last j nodes of the FSM. The dynamic programming algorithm iterates over i in the reversed order. At each step, it tries to either assign the current (s_i, a_i) pair to the current FSM node j, or to create a new transition from another FSM node k to j. We present the detailed algorithm in the supplementary material. Although the transition model we have discussed so far is deterministic, the methods can all be extended straightforwardly to the stochastic case, as also described in the supplement.

To improve the optimization, we add a contrastive loss term, encoding the idea that, for each demonstration $(\bar{s}, \bar{a})$, the corresponding task description t should have a higher rationality score compared to an unmatched task description t', yielding the final objective to be maximized:

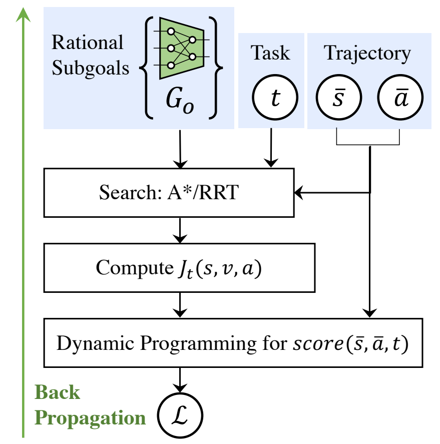

Figure 5: An overview of the training paradigm for RSGs. See text for details.

Algorithm 1: Overview of the training paradigm in pseudocode.

```
Initiate the goal condition $G_o(\cdot;\theta)$
for $(\bar{s}, \bar{a}, t) \in D$ do
    for $t'$ in candidate task descriptions do
        Apply A* search from all states in $\bar{s}$ with task $t'$ to compute a tree $T$.
        for each node $(s, v, a, t') \in T$ in reversed topological order do
            Compute $J_{t'}(s, v, a; \theta)$ on the node using Eq. 4.
        end for
        for each node $(s, v, a, t') \in T$ in reversed topological order do
            Compute Rat $(s, v, a, t'; \theta)$ for each tree node using Eq. 3.
        end for
        Compute $score(\bar{s}, \bar{a}, t'; \theta)$ using Eq. 2 based on Rat values of nodes in $T$.
    end for
    Compute the training objective $\mathcal{J}(\theta)$ using the score of all candidate task descriptions $t'$ using Eq. 5.
    Update $\theta$ using gradient descent by maximizing $\mathcal{J}(\theta)$.
end for
```

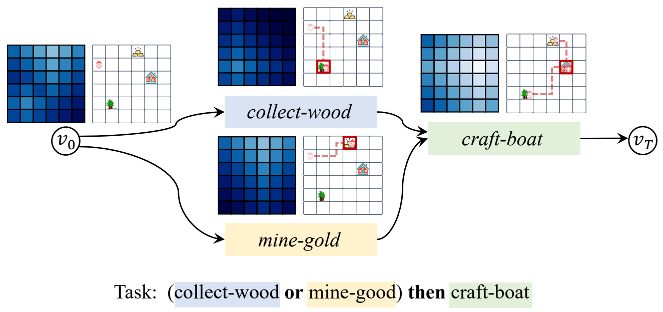

Figure 6: An example of the value function for task-augmented states on a simple FSM. $\min_{a \in \mathcal{A}} J_t(s, v, a)$ are plotted at each location at each FSM node. Deeper color indicates larger cost. Red boxes and dotted lines illustrate the goal and a *rational* trajectory for each subgoal.

$$\mathcal{J}(\theta) = \sum_{(\bar{s},\bar{a},t)\in\mathcal{D}} \Bigg(score(\bar{s}, \bar{a}, t; \theta) + \gamma \cdot \log \frac{\exp(\beta \cdot score(\bar{s}, \bar{a}, t; \theta))}{\sum_{t'} \exp(\beta \cdot score(\bar{s}, \bar{a}, t'; \theta))} \Bigg), \quad (5)$$

where t's are uniformly sampled negative tasks in $\mathcal{TL}$. This loss function is fully differentiable w.r.t. θ, which enables us to apply gradient descent for optimization. Essentially, we are back-propagating through two dynamic programming computation graphs: one that computes J_t based on planning optimal trajectories given goal classifiers parameterized by θ, and one that finds the optimal task-state transitions for the observed trajectory.

RSG Dependency Discovery and Planning

Next, we describe our algorithm for planning with a single, final goal term (e.g., *craft-boat*) instead of step-by-step instructions. Since directly planning for the goal based on the corresponding goal classifier can be very slow due to the long horizon, our key idea here is to leverage the RSGs learned from data to perform a bilevel search. Our algorithm begins with discovering a dependency matrix between RSGs during training time. At performance time, we first use the discovered dependency model to suggest high-level plans, in the form of step-by-step instructions in $\mathcal{TL}$. Next, we use these instructions to plan for environmental actions using our planning algorithm.

For each possible subgoal o, we evaluate the associated learned goal classifier G_o over all states along training trajectories that contain o. Next, we compute *first*$(\bar{s}, o)$ as the smallest index i such that $G_o(s_i)$ is true. If such i does not exist (i.e., G_o is never satisfied in $\bar{s}$) or o is not mentioned in the task specification t associated with $\bar{s}$, we define *first*$(\bar{s}, o) = \infty$. For all tuples $(\bar{s}, o_1, o_2)$, we say o_2 is achieved *before* o_1 if neither *first*$(\bar{s}, o_1)$ nor *first*$(\bar{s}, o_1)$ is infinity, and *first*$(\bar{s}, o_2) <$ *first*$(\bar{s}, o_1)$.

Let *bcount*(o_1, o_2) be the number of $\bar{s} \in \mathcal{D}$ such that o_2 is achieved before o_1 in $\bar{s}$. We construct a dependency matrix d by normalizing the *bcount* as:

$$d(o_1, o_2) \triangleq \frac{bcount(o_1, o_2)}{\sum_{o'} bcount(o_1, o')}, \quad (6)$$

where o' sums over all RSGs.

The derived dependency matrix can be interpreted as the probability that o_2 is a precondition for o_1. Now, recall that our task is to find an action sequence $\bar{a}$ that, starting from the initial state s_0, yields a new state s_T that satisfies the given goal action term g, such as *craft-boat*. Our high-level idea is to leverage the dependency matrix to suggest possible step-by-step instructions t, whose last action term is g. The planning algorithm will follow the suggested instructions to generate low-level plans $\bar{a}$.

Formally, we only consider instructions that are action terms connected by the *then* connective. Denote a candidate instruction $t = o_1$ then o_2 then $\cdots$ then o_k. We define its priority as:

$$\mathit{priority}(t) = \lambda^k \prod_{i=1}^{k-1} \left(1 - \prod_{j=i+1}^{k} (1 - d(o_j, o_i)) \right), \quad (7)$$

where λ is a length bias constant which is set to 0.9 because we prefer shorter instructions.

Given the candidate instructions, we run the planning algorithm for these instructions. We prioritize instructions t with high priorities $\mathit{priority}(t)$, and these instructions are generated by a search approach (Algorithm 2) from the given final goal. The limit of instruction length, *length_limit*, is set to 6 for our experiment.. For more complicated domains, a promising future direction is to learn a full abstract planning model (symbolic or continuous) based on the subgoal terms learned from demonstrations.

Algorithm 2: Overview of the search algorithm given only the final goal.

```
Build a priority queue of instructions $H$.
$H \leftarrow \{\mathit{final\_goal}\}$
while $H$ is not empty do
    $t \leftarrow H.pop()$
    Run A* search on task $t$.
    if the A* search finds a solution then
        Return the solution.
    end if
    if $\mathit{length}(t) \leq \mathit{length\_limit}$ then
        for $o \in O$ do
            if $o \notin t$ and $\exists o' \in t. d(o', o) > 0$ then
                $H.push(o \underline{\mathit{then}}\ t)$  # See Eq. 7.
            end if
        end for
    end if
end while
```

Experiments

We compare our model with other subgoal-learning approaches in Crafting World (Chen, Gupta, and Marino 2021), a Minecraft-inspired crafting environment, and Playroom (Konidaris, Kaelbling, and Lozano-Perez 2018), a 2D continuous domain with geometric constraints.

Crafting World. In Crafting World, the agent can move in a 2D grid world and interact with objects next to it, including picking up tools, mining resources, and crafting items. Mining in the environment typically requires tools, while crafting tools and other objects have their own preconditions, such as being close to a workstation or holding another specific tool. Thus, crafting a single item often takes multiple subgoal steps. There are also obstacles such as rivers (which require boats to go across) and doors (which require specific keys to open).

We define 26 primitive tasks, instantiated from templates of *grab-X*, *toggle-switch*, *mine-X*, and *craft-X*. While generating trajectories, all required items have been placed in the agent's inventory. For example, before mining wood, an axe must be already in the inventory. In this case, the agent is expected to move to a tree and execute the mining action. We also define 26 *compositional* tasks composed of the aforementioned primitive tasks. For each task, we have 400 expert demonstrations.

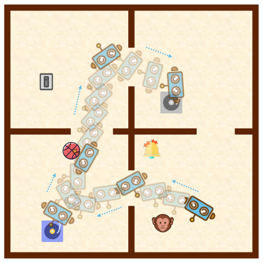
Figure 7: An illustration of the Playroom environment and a trajectory for the task: *turn-on-music* then *play-with-ball* then *turn-off-music*.

All models are trained using tuples of task description t and expert state-action sequences $(\bar{s}, \bar{a})$. In particular, we train all models on primitive and *compositional* tasks and test them on two splits: *compositional* and *novel*. The *compositional* split contains novel state-action sequences of previously-seen tasks. The novel split contains 12 novel tasks, where primitive tasks are composed in ways never seen during training (i.e., not in the 26 tasks from the *compositional* split).

Playroom. Our second environment is Playroom (Konidaris, Kaelbling, and Lozano-Perez 2018), a 2D maze with continuous coordinates and geometric constraints. Fig. 7 shows an illustrative example of the environment. Specifically, a 2D robot can make moves in a small room with obstacles. The agent has three degrees of freedom (DoFs): x and y direction movement, and a 1D rotation. The environment invalidates movements that cause collisions between the agent and the obstacles. Additionally, there are six objects randomly placed in the room, which the robot can interact with. For simplicity, when the agent is close to an object, the corresponding robot-object interaction will be automatically triggered.

Similar to the Crafting World, we have defined six primitive tasks (corresponding to the interaction with six objects in the environment) and eight compositional tasks (e.g., *turn-on-music* then *play-with-ball*). We have designed another eight novel tasks, and for each task, we have 400 expert demonstrations. We train different models on rational demonstrations for both the primitive and compositional tasks, and evaluate them on the compositional and novel splits.

Baselines

We compare our RSGs, which learns goal-based representations, with two baselines using different underlying representations: IRL methods learn reward-based representations, and behavior cloning methods directly learn policies. The implementation details are in the supplementary material.

Our max-entropy inverse reinforcement learning (IRL; Ziebart et al. 2008) baseline learns a task-conditioned reward function by trying to explain the demonstration. For planning, we use the built-in deep-Q-learning algorithm. The behavior cloning (BC; Torabi, Warnell, and Stone 2018) baseline directly learns a task-conditioned policy that maps the

Model	Task Input	Env. Tran.	Crafting World		Playroom	
			Com.	Novel	Com.	Novel
IRL	Lang.	Y	36.5	1.8	28.3	9.6
BC	Lang.	N	11.2	0.8	15.8	4.8
BC-FSM	FSM	N	5.2	0.3	38.2	31.5
RSGs	FSM	Y	**99.6**	**97.8**	**82.0**	**78.2**

Table 1: Results of the planning task, evaluated as the success rate of task completion. IRL and BC take raw task specification and process them with LSTM, while BC-FSM and RSGs uses the FSM directly. RSGs and IRL use the environmental transition model during training while BC and BC-FSM dot not. The maximum number of expanded nodes for all planners is 5,000. All models are trained on the *compositional* split, and tested on the *compositional* and the *novel* split.

current state and the given task to an environment primitive action. BC-FSM is the BC algorithm augmented with our FSM description of tasks. Compared with RSGs, instead of segmenting the demonstration sequence based on rationality, BC-FSM segments them based on how consistent each fragment is with the policy for the corresponding action term.

Results

To evaluate planning, each algorithm is given a new task t, either specified in $\mathcal{TL}$, or as a black-box goal state classifier, and generates a trajectory of actions to complete the task.

Planning with instructions. Table 1 summarizes the results. Overall, RSGs outperforms all baselines. On the *compositional* split, our model achieves a nearly perfect success rate in the Crafting World (99.6%). Comparatively, although the tasks have been presented during training of all baselines, their scores remain below 40%.

On the *novel* split, RSGs outperforms all baselines by a larger margin than on the *compositional* split. We observe that since *novel* tasks contain longer descriptions than those in the *compositional* set, all baselines have a success rate of almost zero. Compared with IRL methods, the more compositional structure in our goal-centric representation allows it to perform better. Meanwhile, a key difference between behavior cloning methods (BC and BC-FSM) and ours is that BC directly applies a learned policy, while our model runs an A* search based on the learned goal classifier and leverages the access to the transition model. This suggests that learning goals is more sample-efficient than learning policies in such domains and generalizes better to new maps.

Our model can be easily applied to environments with image-based states, simply by changing the inputs of I_o and G_o models to images. We evaluate our model in an image-based Crafting World environment. It achieves 82.0% and 78.2% success rates on the compositional and novel splits, respectively. Comparatively, the best baseline BC-FSM gets 38.2% and 31.5%. Details are in the supplementary material.

Planning with goals. We also evaluate RSGs on planning with a single goal action term. These problems require a long solution sequence, making them too difficult to solve with a blind search from an initial state. Since there is no task specification given, in order to solve the problems efficiently, it is critical to use other dependent RSGs for search guidance. We use 8 manually designed goal tests, each of which can be decomposed into 2–5 subgoals. We run our hierarchical search based on RSGs and the discovered dependencies.

We compare this method with two baselines: a blind forward-search algorithm, and a hierarchical search based on RSGs without discovered dependencies (i.e., by setting the dependency matrix as a uniform distribution). We test all three methods on 100 random initial states for each task. Fig. 8 summarizes the result. Overall, RSGs with discovered dependencies enables efficient searches for plans. On easier tasks (2 or 3 subgoals), search with RSGs and dependencies has a similar runtime as the baseline that searches without dependencies. Both of them outperform the blind-search baseline (about 2.4× more efficient when reaching a 70% success rate). However, when the task becomes complex (4 or 5 subgoals), searching with RSGs and the discovered dependencies significantly outperforms other alternatives. For example, to reach a 70% success rate, searching with RSGs needs only 4,311 expanded nodes. By contrast, searching without RSGs needs 19,220 (4.5×) nodes. Interestingly, searching with RSGs but without discovered dependencies performs worse than the blind-search baseline. We hypothesize that this is because it wastes time on planning for unreasonable instructions. Overall, the effectiveness of RSGs with discovered dependencies grows as the complexity of tasks grows.

Related Work

Modular policy learning and planning. Researchers have been learning modular "policies" by simultaneously looking at trajectories and reading task specifications in the form of action term sequences (Corona et al. 2021; Andreas, Klein, and Levine 2017; Andreas and Klein 2015), programs (Sun, Wu, and Lim 2020), and linear temporal logic (LTL) formulas (Bradley et al. 2021; Toro Icarte et al. 2018; Tellex et al. 2011). However, they either require additional annotation for segmenting the sequence and associating fragments with labels in the task description (Corona et al. 2021; Sun, Wu, and Lim 2020), or cannot learn models for planning (Tellex et al. 2011). By contrast, RSGs learns useful subgoals from demonstrations. We use a small but expressive subset of LTL for task description, and jointly learn useful subgoals and segment the demonstration sequence.

Our subgoal representation is also related to other models in domain control knowledge (de la Rosa and McIlraith 2011), goal-centric policy primitives (Park et al. 2020), macro learning (Newton et al. 2007), options and hierarchical reinforcement learning (HRL; Sutton, Precup, and Singh 1999; Dietterich 2000; Barto and Mahadevan 2003; Mehta 2011), and methods that combine reinforcement learning and planning (Segovia-Aguas, Ferrer-Mestres, and Jonsson 2016; Winder et al. 2020). However, the execution of subgoals in RSGs is fundamentally different from options: each option has a policy that we can follow to achieve the short-term goal, while subgoals in RSGs should be refined with segments of primitives by planning algorithms. Our planning algorithm is similar to other approaches: (de la Rosa and McIlraith 2011;

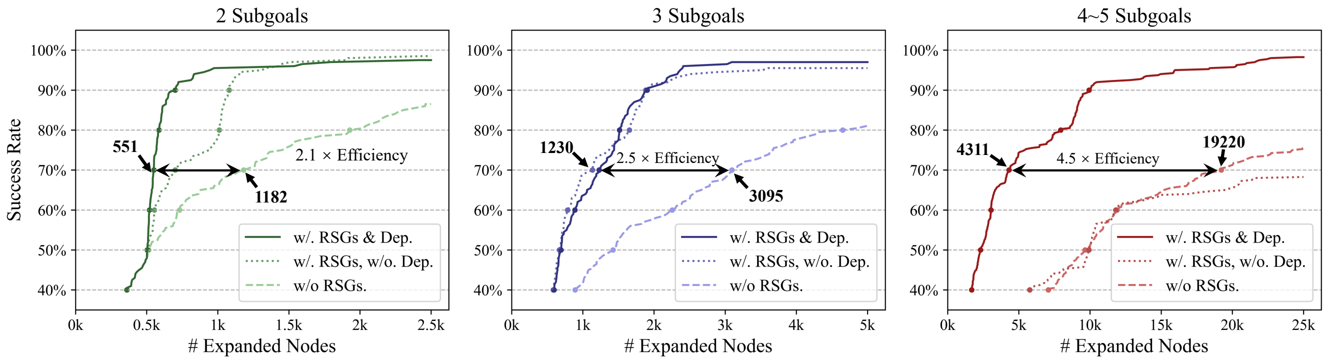

Figure 8: RSGs applied to planning with a final goal. We do evaluation on 3 groups of planning tasks in the Crafting World environment. We use 100 random initial states for each task. Each search method can expand up to 25,000 nodes.

Botvinick and Weinstein 2014; Winder et al. 2020), but they do not leverage discovered dependencies between subgoals.

Learning from demonstration. Learning from demonstration generally refers to building agents that can interact with the environment by observing expert demonstrations (e.g., state-action sequences). Techniques for learning from demonstration can be roughly categorized into four groups: policy function learning (Chernova and Veloso 2007; Torabi, Warnell, and Stone 2018), cost and reward function learning (Markus Wulfmeier and Posner 2015; Ziebart et al. 2008), generative adversarial learning (Ho and Ermon 2016; Liu et al. 2022), and learning high-level plans (Ekvall and Kragic 2008; Konidaris et al. 2012). We refer to Argall et al. (2009) and Ravichandar et al. (2020) as comprehensive surveys. In this paper, we learn useful subgoals that support planning, and compare our model with methods that directly learn policies and cost functions. Moreover, unlike those who use similarities between different actions (Niekum et al. 2012) to segment demonstrations, in RSGs, we segment the demonstration with associate action terms by rationality assumptions of the agent.

Inverse planning. Our model is also related to inverse planning algorithms that infer agent intentions from behavior by finding a task description t that maximizes the consistency between the agent's behavior and the synthesized plan (Baker, Saxe, and Tenenbaum 2009). While existing work has largely focused on modeling the rationality of agents (Baker, Saxe, and Tenenbaum 2009; Zhi-Xuan et al. 2020) and more expressive task descriptions (Shah et al. 2018), our focus is on leveraging the learned subgoals and their dependencies to facilitate agent planning for novel tasks.

Unsupervised subgoal discovery. Our method is also related to approaches for discovering subgoals from unlabelled trajectories (Paul, Vanbaar, and Roy-Chowdhury 2019; Tang et al. 2018; Kipf et al. 2019; Lu et al. 2021; Gopalakrishnan et al. 2021), mostly based on the assumption that the trajectory can be decomposed into segments, and each segment corresponds to a subgoal. Some other approaches for discovering subgoals are to detect "bottleneck" states (Menache, Mannor, and Shimkin 2002; Şimşek, Wolfe, and Barto 2005) based on the state transition graphs. RSG differs from these works in that we focus on learning the grounding of action terms defined in task descriptions. Thus, RSGs are associated with action terms and thus can be recomposed by human users to describe novel tasks. It is a meaningful future direction to combine learning from trajectory-only data and trajectories with descriptions to improve the data efficiency.

Conclusion

We have presented a subgoal learning framework for long-horizon planning tasks. The rational subgoals (RSGs) can be learned by observing expert demonstrations and reading task specifications described in a simple task language $\mathcal{TL}$. Our learning algorithm simultaneously segments the trajectory into fragments corresponding to individual subgoals, and learns planning-compatible models for each subgoal. Our experiments suggest that our framework has strong compositional generalization to novel tasks.

Limitation. The assumption of a deterministic environment has allowed us to focus on the novel RSG formulation of subgoal models. For domains with substantial stochasticity, the high-level concepts of RSGs could be retained (e.g., rationality), and algorithmic changes may be required such as replacing maximum entropy IRL with maximum causal entropy (Ziebart, Bagnell, and Dey 2010). Another limitation of RSGs is that it can not leverage trajectories without labeled task descriptions. Future work may consider the jointly learning of subgoals and subgoal structures of tasks (Vazquez-Chanlatte et al. 2018; Chou, Ozay, and Berenson 2022).

Acknowledgement. We thank Yunyun Wang for giving advice on making figures. We thank all group members of the MIT Learning & Intelligent Systems Group for helpful comments on an early version of the project. This work is in part supported by NSF grant 2214177, AFOSR grant FA9550-22-1-0249, ONR MURI grant N00014-22-1-2740, the MIT-IBM Watson Lab, the MIT Quest for Intelligence, the Center for Brain, Minds, and Machines (CBMM, funded by NSF STC award CCF-1231216), the Stanford Institute for Human-Centered Artificial Intelligence (HAI), and Analog, Amazon, JPMC, Meta, Salesforce, and Samsung. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

Andreas, J.; and Klein, D. 2015. Alignment-Based Compositional Semantics for Instruction Following. In *EMNLP*.

Andreas, J.; Klein, D.; and Levine, S. 2017. Modular multitask reinforcement learning with policy sketches. In *ICML*.

Argall, B. D.; Chernova, S.; Veloso, M.; and Browning, B. 2009. A survey of robot learning from demonstration. *Rob Auton Syst.*, 57(5): 469–483.

Baker, C. L.; Saxe, R.; and Tenenbaum, J. B. 2009. Action understanding as inverse planning. *Cognition*, 113(3): 329–349.

Barto, A. G.; and Mahadevan, S. 2003. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1): 41–77.

Botvinick, M.; and Weinstein, A. 2014. Model-based hierarchical reinforcement learning and human action control. *Philos. Trans. R. Soc. Lond., B, Biol. Sci.*, 369(1655): 20130480.

Bradley, C.; Pacheck, A.; Stein, G. J.; Castro, S.; Kress-Gazit, H.; and Roy, N. 2021. Learning and Planning for Temporally Extended Tasks in Unknown Environments. arXiv:2104.10636.

Chen, V.; Gupta, A.; and Marino, K. 2021. Ask Your Humans: Using Human Instructions to Improve Generalization in Reinforcement Learning. In *ICLR*.

Chernova, S.; and Veloso, M. 2007. Confidence-based policy learning from demonstration using gaussian mixture models. In *AAMAS*.

Chou, G.; Ozay, N.; and Berenson, D. 2022. Learning temporal logic formulas from suboptimal demonstrations: theory and experiments. *Autonomous Robots*, 46(1): 149–174.

Corona, R.; Fried, D.; Devin, C.; Klein, D.; and Darrell, T. 2021. Modular Networks for Compositional Instruction Following. In *NAACL-HLT*, 1033–1040.

De Giacomo, G.; and Vardi, M. Y. 2013. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*.

de la Rosa, T.; and McIlraith, S. 2011. Learning domain control knowledge for TLPlan and beyond. In *ICAPS 2011 Workshop on Planning and Learning*.

Dietterich, T. G. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *JAIR*, 13: 227–303.

Dong, H.; Mao, J.; Lin, T.; Wang, C.; Li, L.; and Zhou, D. 2019. Neural Logic Machines. In *ICLR*.

Ekvall, S.; and Kragic, D. 2008. Robot learning from demonstration: a task-level planning approach. *IJARS*, 5(3): 33.

Gopalakrishnan, A.; Irie, K.; Schmidhuber, J.; and van Steenkiste, S. 2021. Unsupervised Learning of Temporal Abstractions using Slot-based Transformers. In *Deep RL Workshop at NeurIPS*.

Ho, J.; and Ermon, S. 2016. Generative adversarial imitation learning. In *NeurIPS*.

Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural Comput.*, 9(8): 1735–1780.

Kipf, T.; Li, Y.; Dai, H.; Zambaldi, V.; Sanchez-Gonzalez, A.; Grefenstette, E.; Kohli, P.; and Battaglia, P. 2019. Compile: Compositional imitation learning and execution. In *ICML*.

Konidaris, G.; Kaelbling, L. P.; and Lozano-Perez, T. 2018. From skills to symbols: Learning symbolic representations for abstract high-level planning. *JAIR*, 61: 215–289.

Konidaris, G.; Kuindersma, S.; Grupen, R.; and Barto, A. 2012. Robot learning from demonstration by constructing skill trees. *IJRR*, 31(3): 360–375.

LaValle, S. M.; et al. 1998. Rapidly-exploring random trees: A new tool for path planning. Technical report, Computer Science Department, Iowa State University.

Liu, M.; Zhu, Z.; Zhuang, Y.; Zhang, W.; Hao, J.; Yu, Y.; and Wang, J. 2022. Plan Your Target and Learn Your Skills: Transferable State-Only Imitation Learning via Decoupled Policy Optimization. In *ICML*.

Lu, Y.; Shen, Y.; Zhou, S.; Courville, A.; Tenenbaum, J. B.; and Gan, C. 2021. Learning task decomposition with ordered memory policy network. In *ICLR*.

Markus Wulfmeier, P. O.; and Posner, I. 2015. Maximum Entropy Deep Inverse Reinforcement Learning. In *NeurIPS Workshop*.

Mehta, N. 2011. *Hierarchical structure discovery and transfer in sequential decision problems*. Oregon State University.

Menache, I.; Mannor, S.; and Shimkin, N. 2002. Q-cut—dynamic discovery of sub-goals in reinforcement learning. In *European conference on machine learning*, 295–306. Springer.

Newton, M. A. H.; Levine, J.; Fox, M.; and Long, D. 2007. Learning Macro-Actions for Arbitrary Planners and Domains. In *ICAPS*.

Niekum, S.; Osentoski, S.; Konidaris, G.; and Barto, A. G. 2012. Learning and generalization of complex tasks from unstructured demonstrations. In *IROS*. IEEE.

Park, D.; Noseworthy, M.; Paul, R.; Roy, S.; and Roy, N. 2020. Inferring task goals and constraints using bayesian nonparametric inverse reinforcement learning. In *JMLR*.

Paul, S.; Vanbaar, J.; and Roy-Chowdhury, A. 2019. Learning from trajectories via subgoal discovery. *Advances in Neural Information Processing Systems*, 32.

Ravichandar, H.; Polydoros, A. S.; Chernova, S.; and Billard, A. 2020. Recent advances in robot learning from demonstration. *Annu Rev Control.*, 3: 297–330.

Segovia-Aguas, J.; Ferrer-Mestres, J.; and Jonsson, A. 2016. Planning with partially specified behaviors. In *Artificial Intelligence Research and Development*, 263–272. IOS Press.

Shah, A.; Kamath, P.; Shah, J. A.; and Li, S. 2018. Bayesian Inference of Temporal Task Specifications from Demonstrations. In *NeurIPS*.

Şimşek, Ö.; Wolfe, A. P.; and Barto, A. G. 2005. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd international conference on Machine learning*, 816–823.

Sun, S.-H.; Wu, T.-L.; and Lim, J. J. 2020. Program guided agent. In *ICLR*.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2): 181–211.

Tang, D.; Li, X.; Gao, J.; Wang, C.; Li, L.; and Jebara, T. 2018. Subgoal discovery for hierarchical dialogue policy learning. *arXiv preprint arXiv:1804.07855*.

Tellex, S.; Kollar, T.; Dickerson, S.; Walter, M.; Banerjee, A.; Teller, S.; and Roy, N. 2011. Understanding natural language commands for robotic navigation and mobile manipulation. In *AAAI*.

Torabi, F.; Warnell, G.; and Stone, P. 2018. Behavioral Cloning from Observation. In *IJCAI*.

Toro Icarte, R.; Klassen, T. Q.; Valenzano, R.; and McIlraith, S. A. 2018. Teaching Multiple Tasks to an RL Agent Using LTL. In *AAMAS*.

Vazquez-Chanlatte, M.; Jha, S.; Tiwari, A.; Ho, M. K.; and Seshia, S. 2018. Learning task specifications from demonstrations. In *NeurIPS*.

Winder, J.; Milani, S.; Landen, M.; Oh, E.; Parr, S.; Squire, S.; Matuszek, C.; et al. 2020. Planning with abstract learned models while learning transferable subtasks. In *AAAI*.

Zhi-Xuan, T.; Mann, J. L.; Silver, T.; Tenenbaum, J. B.; and Mansinghka, V. K. 2020. Online bayesian goal inference for boundedly-rational planning agents. In *NeurIPS*.

Ziebart, B. D.; Bagnell, J. A.; and Dey, A. K. 2010. Modeling interaction via the principle of maximum causal entropy. In *ICML*.

Ziebart, B. D.; Maas, A. L.; Bagnell, J. A.; and Dey, A. K. 2008. Maximum entropy inverse reinforcement learning. In *AAAI*.

Supplementary Material for Learning Rational Subgoals from Demonstrations and Instructions

First, we elaborate how A^* search is performed on the FSM-augmented transition models. We also discuss representation choices of RSGs, as well as the optimality, complexity and scalability of the search algorithms. Recall that we are using dynamic programming obtain deterministic transitions, and there are other formulations such as using stochastic transitions, we talk about the comparison between our formulation and others in this section. In addition, we provide details for the dependency discovering algorithm and hierarchical search algorithm used in planning for final goals.

Second, we discuss details about datasets and how we process the data, including how the features are extracted from the state representations. We also provide the list of task descriptions covered in each data split.

Next, we provide implementation details for baselines, and then discusses the limitation and future work of RSGs.

Implementation Details of RSGs

Re-parameterize $1 - G_o(\cdot)$ using a separate neural network. In practice, instead of directly using $1 - G_o(\cdot)$ to evaluate the probability that a subgoal has not been met, we parameterize $1 - G_o(\cdot)$ using a separate neural network $I_o(\cdot)$ that has the same architecture as $G_o(\cdot)$. We observe that this re-parameterization stabilizes the training. In performance time, we will only be using the goal classifier G_o and ignores the I_o.

Empirically, we find that when using a single subgoal classifier instead of separate classifiers for I and G, some classifiers usually get stuck at local optima. As a result, the overall planning performance, on average, drops from 99.6% to 75%. We hypothesize that this is because using separate parameterization allows a broader set of possible solutions to the original problem, which are practically equivalently helpful for planning. As a concrete example, consider the subgoal "mining-wood."

If we use only the G and $1 - G$ parameterization, the only feasible solution is:

$$G_1 = [\text{wood in inventory}]$$

However, if we use separate parameterizations, the following solution will also be accepted:

$$I_2 = [\text{tree on map and axe in inventory and wood not in inventory}]$$

$$G_2 = [\text{tree on map and axe in inventory and wood in inventory}]$$

Note that, during planning, both classifiers G_1 and G_2 have the same effects, but the relaxed parameterization allows a broader set of solutions.

FSM-A^*

We have implemented a extended version of the A^* algorithm to handle FSM states in Crafting World.

A^* at each FSM node. We start with the A^* search process happening at each FSM node. For a given FSM state, the A^* search extends the tree search in two stages. The first stage lasts for $b = 3$ layers during training and $b = 4$ layers during testing. In the first b layers of the search tree, we run a Breadth-First-Search so that every possible path with length b is explored. Then on the second stage lasts for $c = 15$ layers during training and 25 layers in testing. In layer $d \in [b + 1, b + c]$, we run A* from the leaves in the first stage based on the heuristic for each node. By enforcing the exploration at the early stage, we avoid imperfect heuristic from misguiding the A* search at the beginning. For each FSM node v and each layer d, we only keep the top $k = 10$. Finally, we run the value iteration on the entire search tree.

To accelerate this search process, for all tasks t in the training set, we have initialized a dedicated value approximator $V_t(\bar{s})$, conditioned on the historical state sequence. During training, we use the value iteration result on the generated search tree to supervise the learning of this approximator V_t. Meanwhile, we use the value prediction of V_t as the heuristic function for node pruning. During test, since we may encounter unseen tasks, the A^*-FSM search uses a uniform heuristic function $h \equiv 0$.

Search on an FSM. For a given initial state s_0 and task description t, we first build FSM$_t$ and add the search tree node (s_0, v_0) to the search tree, where v_0 is the initial node of the FSM. Then we expand the search tree nodes (s, v) by a topological order of v. It has two stages. First, for each FSM node v, we run up to 5000 steps of A^* search. Next, for all search tree nodes (s, v) at FSM node v, we try to make a transition from (s, v) to (s, v') where (v, v') is a valid edge in FSM$_t$. Finally, we output a trajectory ending at the FSM node v_T with minimum cost.

Optimality of the A* algorithm on FSM. In the current implementation, RSGs might return sub-optimal solutions even with a perfect heuristic, because RSGs balance the expanded nodes across all FSM nodes: it first samples an FSM node and then expand a search tree node with the best heuristic value on that node.

The optimality can be guaranteed by either of the following simple modifications, although at the cost of possibly increasing the running time:

- Always expand the search node with the globally best admissible heuristic value. (Because our heuristic is learned, this may not be practical.)
- Keep expanding nodes, even after finding a plan, until none of the unexpanded search tree nodes across all subgoal nodes in the FSM have better heuristic values than the current best solution.

Transitions on FSM

Algorithm 3: The dynamic programming for computing $score(\bar{s}, \bar{a}, t; \theta)$ given $G_o(\cdot; \theta)$ and Rat$(s_i, v_i, a_i, t; \theta)$.

```
Initialize $f[1..n, 0..T]$ to $-\infty$
Topological sort all FSM nodes $v_{0..T}$ so that for all $0 \le i < j \le T$, there is no path from $v_j$ to $v_i$ on FSM. Clearly $v_0$ is still the start node and $v_T$ is the terminate node.
$f[n, T] \leftarrow 0$
for $i = n..1$ do
    for $j = T..0$ do
        if $i < n$ then
            $f[i, j] \leftarrow \text{Rat}(s_i, v_j, a_i, t; \theta) + f[i+1, j]$
        end if
        for do$k = 0..T$:
            if $(v_j, v_k) \in$ FSM transitions then:
                $f[i, j] \leftarrow \max f[i, j], \log G_{v_j}(s_i; \theta) + \log(1 - G_{v_k}(s_i; \theta) + f[i+1, k]$
            end if
        end for
    end for
end for
Return $score(\bar{s}, \bar{a}, t; \theta) = f[1, 0]$
```

When encoding transitions on FSM, we use dynamic programming [†] to select a transition that maximize our $score$. Algorithm 3 shows the pseudo-code of the dynamic programming. If we consider I_v and G_v as "soft" probabilities, the computation of $score$ finds $\bar{s}'$ and $\bar{a}'$ that maximize the rationality of primitive actions and the likelihood that FSM transitions are successful. We use $score(\bar{s}, \bar{a}, t)$ to rank all candidate tasks.

There is another formulation which is to consider the stochastic transitions using G_v as probabilities. These two formulations can be unified using a framework of latent transition models, though they are computed using different DP algorithms and may lead to different results.

First of all, these two formulations are equivalent when the goal classifiers are binary (0/1). When the classifiers are approximated by "soft" functions that indicate the probability they are satisfied, the two formulations correspond to two approaches of integrating reward (i.e. rationality in our model). The stochastic transition formulation computes the expected rationality, and our formulation can be viewed as an approximation of maximum-likelihood estimated rationality – we take $\max_\tau \lambda \log \Pr(\text{transitions in } \tau \text{ are successful}) + Rationality(\bar{s}, \bar{a}, \tau)$. It would be an interesting extension to adopt the stochastic transition formulation (i.e., $\mathbb{E}_\tau \lambda \log \Pr(\text{transitions in } \tau \text{ are successful}) + Rationality(\bar{s}, \bar{a}, \tau))$ and use a stochastic planner or MDP solver, although the planning time might be substantially increased.

Second, even if these two approaches behave differently in some cases, but it is unclear which one is better: this is a fundamental challenge in planning: how should the robot decide whether it has finished a task if there is no indication (such as rewards) from the environment?

Discover and search with dependencies

Evaluate goal classifiers when computing *first*$(\bar{s}, o)$ for discovering dependencies. Recall that when discovering the dependencies, we need to compute *first*$(\bar{s}, o)$, the smallest index i such that $G_o(s_i)$ is true. We need to round $G_o(s_i)$ to Boolean value because we internally use the soft version of $G_o(\cdot)$.

For each subgoal o, we evaluate G_o on all states in the training set to obtain the minimum and maximum output $MinG_o$ and $MaxG_o$, and we consider $G_o(s_i)$ to evaluate to true iff $G_o(s_i) \ge \sqrt{MinG_o \cdot MaxG_o}$.

[†]The dynamic programming is similar to Dynamic time warping(DTW) warping trajectories into sequential subgoals, but the "cost" is computed at each segment instead of at matched positions.

Tuning hyperparameters. There are several hyper-parameters that need to be tuned: $\lambda, \alpha, \gamma, \beta$, and we discuss them separately below:

- λ is determines the importance of satisfying the transition condition when transiting compared to regular action cost. In the environment, the cost of a single move $\mathcal{C}(s, a)$ is set to 0.1, and λ is set to 1.
- α is called the inverse rationality, the higher α, the more rationality is assumed of the agent i.e. the agent is assume to take the optimal action with higher strategy. We set $\alpha = 1$ in our experiments.
- λ, α and the environmental action costs jointly determine the weight of FSM transitions, as well as the tolerance of suboptimality in the demonstrations (the higher α the lower tolerance). Our model is not sensitive to α (setting $0.1 < \alpha < 10$ have similar performance) because the trajectories in our dataset are close to optimal, and our model is a bit sensitive to the ratio of λ to the action cost. We found that expected-number-of-steps-per-subgoal $\times$ action-cost is a good value for λ—it gives about the same weight to the transition appropriateness and the rationality along the path to achieve it.
- γ is the weight of the classification loss in the loss function $\mathcal{J}(\theta)$. We set $\gamma = 0.1$ and we found that $0.01 < \gamma < 0.1$ all work well.

  β is to adjust the base for the softmax function for the classification, which is set to 1. Note that β and γ jointly determine the weight of classification loss. Similarly, we have found that $0.1 < \beta < 1$ all work well.

Scalability and complexity of instruction generating. Meanwhile, the efficiency of our hierarchical search can be justified theoretically, even at the worst case when we are doing enumerating search approach without well-discovered dependencies. Say we have a subgoal set $\mathcal{O}$, a primitive action set $\mathcal{A}$, and each subgoal can be completed in l actions, and the task can be achieved by sequencing m subgoals. Our two-level search generates up to $O(|\mathcal{O}|^m)$ candidate subgoal sequences and searching each sequence takes $O(m|\mathcal{A}|^l)$ time. Thus, the worst-case complexity is $O(m|\mathcal{O}|^m|\mathcal{A}|^l)$ which is still better than a pure primitive-level search, which is at worst $O(|\mathcal{A}|^{ml})$, because the number of subgoals $|\mathcal{O}|$ is usually much smaller than $|\mathcal{A}|^l$ (the number of all possible length l sequences).

Dataset

Crafting World

Our Crafting World environment is based on the Crafting environment introduced by (Chen, Gupta, and Marino 2021). The environment has a crafting agent that can move in a grid world, collect resources, and craft items. In the original environment, every crafting rule is associated with a unique crafting station (e.g., paper must be crafted on the paper station). We modified the rules such that some crafting rules can share a common crafting station (e.g., both arrows and swords can be crafted on a weapon station). We add additional tiles: doors and rivers into the environment. Toggling a specific switch will open all doors. Otherwise, the agent can move across doors when they are holding a key. Meanwhile, the agent can move across rivers when they have a boat in their inventory.

We have used 47 object types in Crafting World including obstacles (e.g., river tiles, doors), items (e.g., axe), resources (e.g., trees), and crafting stations. We use 27 rules for mining resources and crafting items. When the agent is at the same tile as another object, the *toggle* action will trigger the object-specific interaction. For item, the *toggle* action will pick up the object. For resource, the *toggle* action will mine the resource if the agent has space in their inventory and has the required tool for mining this type of resource (e.g., pickaxe is needed for mining iron ore).

State representation. The state representation of Crafting World consists of three parts.

1. The *global feature* contains the size of grid world, the location of the agent, and the inventory size of the agent.
2. The *inventory* feature contains an unordered list of objects in the agent's inventory. Each of them is represented as a one-hot vector indicating its object type.
3. The *map* feature contains all objects on the map, including obstacles, items, resources, and crafting stations. Each of them is represented by a one-hot type encoding, the location (as integer values), and state (e.g., *open* or *closed* for doors).

Action. In Crafting World, there are 5 primitive level actions: *up*, *down*, *left*, *right*, and *toggle*. The first four actions will move the agent in the environment, while the *toggle* action will try to interact with the object in the same cell as the agent.

State feature extractor. Since our state representation contain a varying number of objects, we extract a vector representation of the environment with a relational neural network: Neural Logic Machines (Dong et al. 2019).

Concretely, we extract the inventory feature and the map feature separately. For each item in the inventory, we concatenate its input representation (i.e., the object type) with the global input feature. We process each item with the same fully-connected layer with ReLU activation. Following NLM (Dong et al. 2019), we use a max pooling operation to aggregate the feature for all inventory objects, resulting in a 128-dim vector. We use a similar architecture (but different neural network weights) to process all objects on the map. Finally, we concatenate the extracted inventory feature (128-dim), the map feature (128-dim), and the global feature (4-dim) as the holistic state representation. Thus, the output feature dimension for each state is 260.

Primitive			
grab-pickaxe	grab-axe	grab-key	toggle-switch
craft-wood-plank	craft-stick	craft-shears	craft-bed
craft-boat	craft-sword	craft-arrow	craft-cooked-potato
craft-iron-ingot	craft-gold-ingot	craft-bowl	craft-beetroot-soup
craft-paper	mine-gold-ore	mine-iron-ore	mine-sugar-cane
mine-coal	mine-wood	mine-feather	mine-wool
mine-potato	mine-beetroot		

Compositional	
grab-pickaxe	grab-axe
grab-key	toggle-switch
mine-wood *then* craft-wood-plank	craft-wood-plank *then* craft-stick
craft-iron-ingot *or* craft-gold-ingot *then* craft-shears	mine-wool *and* craft-wood-plank *then* craft-bed
craft-wood-plank *then* craft-boat	craft-iron-ingot *and* craft-stick *then* craft-sword
mine-feather *and* craft-stick *then* craft-arrow	mine-potato *and* mine-coal *then* craft-cooked-potato
mine-iron-ore *and* mine-coal *then* craft-iron-ingot	mine-gold-ore *and* mine-coal *then* craft-gold-ingot
craft-wood-plank *or* craft-iron-ingot *then* craft-bowl	craft-bowl *and* mine-beetroot *then* craft-beetroot-soup
mine-sugar-cane *then* craft-paper	grab-pickaxe *then* mine-gold-ore
grab-pickaxe *then* mine-iron-ore	grab-pickaxe *or* grab-axe *then* mine-sugar-cane
grab-pickaxe *then* mine-coal	grab-axe *then* mine-wood
craft-sword *then* mine-feather	craft-shears *or* craft-sword *then* mine-wool
grab-axe *or* mine-coal *then* mine-potato	grab-axe *or* grab-pickaxe *then* mine-beetroot

Novel
1. mine-sugar-cane *then* craft-paper
2. mine-potato *and* (gran pickaxe then mine-coal) *and* craft-cooked-potato
3. mine-beetroot *and* (grab-axe then mine-wood then craft-wood-plank then craft-bowl) *then* craft-beetroot-soup
4. grab-axe *then* mine-wood *then* craft-wood-plank *then* grab-pickaxe *then* mine-iron-ore *and* mine-coal *then* craft-iron-ingot *then* craft-shears *then* mine-wool *then* craft-bed
5. grab-axe *then* mine-wood *then* craft-wood-plank *then* craft-stick *then* grab-pickaxe *then* mine-iron-ore *and* mine-coal *then* craft-iron-ingot *then* craft-sword *then* mine-feather *then* mine-wood *then* craft-wood-plank *then* craft-stick *then* craft-arrow
6. grab-key *then* grab-axe
7. toggle-switch *then* mine-beetroot
8. grab-axe *then* mine-wood *then* craft-wood-plank *then* craft-boat *then* mine-sugar-cane
9. grab-axe *then* mine-wood *then* craft-wood-plank *then* craft-boat *then* grab-pickaxe
10. grab-key *then* grab-axe *then* mine-wood *then* craft-wood-plank *then* craft-boat *then* mine-potato
11. grab-key *or* (grab-axe *then* mine-wood *then* craft-wood-plank *then* craft-boat) *then* grab-pickaxe *then* mine-gold-ore
12. grab-axe *then* mine-wood *then* craft-wood-plank *then* craft-boat *then* grab-key *or* toggle-switch *then* grab-pickaxe *then* mine-iron-ore *and* mine-coal *then* craft-iron-ingot

Table 2: Task descriptions in the *primitive*, *compositional* and *novel* sets for the Crafting World.

Task definitions. We list the task descriptions in the *primitive*, the *compositional*, and the *novel* splits Table 2. Table 3 lists 8 final goals and corresponding full instructions that are used in search for a single final goal.

Playroom

We build our Playroom environment following Konidaris et al. (Konidaris, Kaelbling, and Lozano-Perez 2018). Specifically, we have added obstacles into the environment. The environment contains an agent, 6 effectors (a ball, a bell, a light switch, a button to turn on the music, a button to turn off the music and a monkey), and a fix number of obstacles. The agent and the effectors have fixed shapes. Thus, their geometry can be fully specified by their location and orientation. For simplicity, we have also fixed the shape and the location of the obstacles.

State representation. We represent the pose of the agent by a 3D vector including the x, y coordinates (real-valued) and its rotation (real-valued, in $[-\pi, \pi)$. The state representation consist of the pose of the agent (as a 3-dimensional vector) and the locations of six effectors (as 6 2-dimensional vectors). Note that the state representation does not contain the shapes nor the locations of obstacles as they remain unchanged throughout the experiment. We concatenate these 7 vectors as the state representation.

Final Goal	Steps	Example full instruction
mine-wood	2	grab-axe *then* mine-wood
craft-paper	2	mine-sugar-cane *then* craft-paper
craft-beetroot-soup	3	(mine-beetroot *and* craft-bowl) *then* craft-beetroot-soup
craft-bed	3	(craft-wood-plank *and* mine-wool) *then* craft-bed
craft-gold-ingot	4	grab-pickaxe *then* (mine-gold-ore *and* mine-coal) *then* craft-gold-ingot
craft-boat	4	grab-axe *then* mine-wood *then* craft-wood-plank *then* craft-boat
craft-cooked-potato	4	((grab-pickaxe *then* mine-coal) *and* mine-potato) *then* craft-cooked-potato
craft-shears	5	grab-pickaxe *then* (mine-coal *and* mine-iron-ore) *then* craft-iron-ingot *then* craft-shears

Table 3: Task descriptions used in search for a single final goal in Crafting World.

Action. The agent has a 3-dimensional action space: $[-1, 1]^3$. That is, for example, at each time step, the agent can at most move 1 meter along the x axis. We perform collision checking when the agent is trying to make a movement. If an action will result in a collision with objects or obstacles in the environment, the action will be treated as invalid and the state of the agent will not change.

Task definitions. We list the task descriptions in each of the *primitive*, *compositional* and *novel* set of the Playroom in Table 4

Primitive		
play-with-ball	ring-bell	turn-on-light
touch the mounkey	turn-off-music	turn-on-music
Compositional (designed meaningful tasks)		
play-with-ball		
turn-on-light *then* ring-bell		
turn-on-music *and* play-with-ball *then* touch the monkey		
play-with-ball *then* turn-on-light		
turn-on-music *and* play-with-ball *then* turn-off-music		
turn-on-music *or* play-with-ball		
turn-off-music *then* play-with-ball *then* turn-on-music		
turn-on-music *and* play-with-ball *and* turn-on-light *then* ring-bell		
Novel (randomly sampled)		
play-with-ball *then* turn-on-light *or* ring-bell		
turn-on-music *then* turn-on-light		
turn-on-music *then* turn-on-light		
play-with-ball *then* touch the monkey		
turn-on-music *then* turn-off-music		
turn-on-music *and* ring-bell *then* touch the monkey		
ring-bell *then* touch the monkey *then* turn-on-light		
turn-on-light *and* (ring-bell *or* turn-on-music) *then* play-with-ball		

Table 4: Task descriptions in the *primitive*, *compositional* and *novel* sets for the Playroom.

Baseline Implementation Details

In this section, we present the implementation details of RSGs and other baselines. Without further notes, through out this section, we will be using the same LSTM encoder for task descriptions in $\mathcal{TL}$, and the same LSTM encoder for state sequences. The architecture of both encoders will be presented in Appendix .

LSTM

Task description encoder. We use a bi-directional LSTM (Hochreiter and Schmidhuber 1997) with a hidden dimension of 128 to encode the task description. The vocabulary contains all primitive subgoals, parentheses, and three connectives (*and*, *or*, and *then*). We perform an average pooling on the encoded feature for both directions, and concatenate them as the encoding for the task description. Thus, the output dimension is 256.

State sequence encoder. For a given state sequence $\bar{s} = \{s_i\}$, we first use a fully-connected layer to map each state s_i into a 128-dimensional vector. Next, we feed the sequence into a bi-directional LSTM module. The hidden dimension of the LSTM is 128. We perform an average pooling on the encoded feature for both directions, and concatenate them as the encoding for the state sequence.

Training. In our LSTM baseline for task recognition, we concatenate the state sequence feature and the task description feature, and use a 2-layer multi-layer perceptron (MLP) to compute the score of the input tuple: (trajectory, task description). The LSTM model is trained for 100 epochs on both environments. Each epoch contains 30 training batches that are randomly sampled from training data. The batch size is 32. We use the RMSProp optimizer with a learning rate decay from 10^{-3} to 10^{-5}.

Inverse Reinforcement Learning (IRL)

The IRL baseline uses an LSTM model to encode task descriptions. We use different parameterizations for the reward function and the Q function in two datasets.

Crafting World Since the task description may have complex temporal structures, the reward value does not only condition on the current state and but all historical states. Therefore, instead of $Q(s, a|t)$ and $R(s, a, s'|t)$, we use $Q(\bar{s}, a|t)$ and $R(\bar{s}, a, s'|t)$ to parameterize the Q function and reward function, where s is the current state, a the action, t the task description, s' the next state, and $\bar{s}$ the historical state sequence from the initial state to the current state.

We use neural networks to approximate the Q function and reward function. For both of them, $\bar{s}$ is first encoded by an LSTM model into a fixed-length vector embedding. We simply concatenate the historical state encoding and the task description encoding, and then use a fully-connected layer to map the feature into a 5-dimensional vector. Each entry corresponds to the Q value or the reward value for a primitive action.

Playroom The Q function and reward function in Playroom also condition on all historical states. In Playroom, we parameterize the value of each state: $V(\bar{s})$, instead of $Q(\bar{s}, a)$. We parameterize $R(\bar{s}, a, s')$ as $R(\bar{s}, s')$.

The input to our reward function network is composed of three parts: the vector encoding of the historical state sequence, the vector encoding for the next state s', and the task description encoding. We concatenate all three vectors and run a fully-connected layer with a logSigmoid activation function.[‡]

In Playroom, since we do not directly parameterize the Q value for all actions in the continuous action space, in order to sample the best action at each state s for plan execution, we first randomly sample 20 valid actions from the action space (i.e., actions that do not lead to collision), and choose the action that maximizes the Q function: $Q(\bar{s} \cup s', a)$, where $\bar{s}$ is the historical state seuqnce and s' is the next state after taking a.

Value iteration. Both environments have a very large (Crafting World) or even infinite (Playroom) state space. Thus it is impossible to run value iteration on the entire state space. Thus, at each iteration, for a given demonstration trajectory $(\bar{s}_e, \bar{a}_e)$, we construct a self exploration trajectory $(\bar{s}_p, \bar{a}_p)$ that share the same start state as $\bar{s}_e$ [§]. We run value iteration on $\{\bar{s}_e\} \cup \{\bar{s}_p\}$. For states not in this set, we use the Q function network to approximate their values.

Training. For both Crafting World and Playroom, we train the IRL model for 60 epochs. We set the batch size to be 32 and each epoch has 30 training batches. We use a replay buffer that can store 100,000 trajectories. For both environments, we use the Adam optimizer with a learning rate decay from 10^{-3} to 10^{-5}. We have found the IRL method unstable to train in the Playroom environment. Thus, in Playroom, we use a warm-up training procedure. In the first 18(30%) epochs, we set $\gamma = 0$ for a "warm start", and for rest of the epochs we use $\gamma = 0.5$, where γ is the discount factor in the Q function.

[‡]We have experimented with no activation function, Sigmoid, and logSigmoid activations, and found that the logSigmoid activation works the best.

[§]Since running self-exploration in Playroom is too slow, in practice, we only generate self-exploration trajectories for 4 trajectories in the input batch.

Model	#Epochs	Crafting World		Playroom	
		Training Time (minute/epoch)	Planning Time (second/sample)	Training Time (minute/epoch)	Planning Time (second/sample)
RSG	60	17.0	4.6962	32.4	0.4313
LSTM	200	0.4	N/A	0.2	N/A
IRL	60	34.8	0.7269	7.1	23.0187
BC	150	0.3	0.1615	0.4	0.1688
BC(FSM)	150	0.5	0.5423	0.5	0.1875

Table 5: Training time per epoch and planning time per sample for all models.

Behavior Cloning (BC)

BC learns a policy $\pi(\bar{s}, a|t)$ from data, where t is the task description, a a primitive action, and $\bar{s}$ the historical state sequence. The state sequence $\bar{s}$ is first encoded by an LSTM model into a fixed-length vector embedding.

In Crafting World, we use a fully-connected layer with softmax activation to parameterize $\pi(a|\bar{s}, t)$. Specifically, the input to the fully-connected layer is the concatenation of the vector encoding of $\bar{s}$ and the vector encoding of the task description t.

In Playroom, we use two fully-connected (FC) layers to parameterize $\pi(a|\bar{s}, t)$. Specifically, we parameterize $\pi(a|\bar{s}, t)$ as a Gaussian distribution. The first FC layer has a Tanh activation and parameterizes the mean μ of the Gaussian. The second FC layer has a Sigmoid activation and paramerizes the standard variance σ^2 of the Gaussian.

To make this model more consistent with our BC-FSM model, in both environments, we also train a module to compute the termination condition of the trajectory. That is, a neural network that maps $\bar{s}$ to a real value in $[0, 1]$, indicating the probability of terminating the execution. Denote the output of this network as $\textit{stop}(\bar{s})$. At each time step, the agent will terminate its policy with probability $\textit{stop}(\bar{s})$. We modulate the probability for other actions a as $\pi(a|\bar{s}, t) \cdot (1 - \textit{stop}(\bar{s}))$

For planning in Crafting World, at each step, we choose the action with the maximum probability (including the option to "terminate" the execution). In Playroom, we always take the "mean" action parameterized by $\pi(a|\bar{s}, t)$ until we reach the maximum allowed steps.

We then define the score of a task given a trajectory, $\textit{score}(\bar{s}, \bar{a}, t)$, as the sum of log-probabilities of the actions taken at each step. We train this model with the same loss and training procedure as RSGs. We train the model for 100 epochs using the Adam optimizer with a learning rate decay from 10^{-3} to 10^{-5}.

Behavior Cloning with FSM (BC-FSM)

BC-FSM represents task description as an FSM, in the same way as our model RSGs. It represents each subgoal o as a tuple: $\langle \pi_o(sa), \textit{stop}_o(s) \rangle$, corresponding to the subgoal-conditioned policy and the termination condition.

Task recognition. The task recognition procedure for BC-FSM jointly segments the trajectory and computes the consistency score between the task description and the input trajectory. In particular, our algorithm will assign an FSM state v_i to each state s_i, and insert several action. We use a dynamic programming algorithm (similar to the one used by our algorithm for RSGs) to find the assignment that maximize the overall score:

$$\textit{score}(\bar{s}, \bar{v}, \bar{a}) \coloneqq \prod_i p(a_i|s_i, v_i, t)$$

$$p(a_i|s_i, v_i, t) = \begin{cases} \pi(a_i|s_i, v_i) \cdot (1 - \textit{stop}(s_i, v_i)) & \text{if } a \in \mathcal{A} \text{ is a primitive action} \\ \textit{stop}(s_i, v_i) & \text{if } a \in E_t \text{ is an FSM transition} \end{cases}$$

Planning. We use the same strategy as the basic Behavior Cloning model to choose actions at each step, conditioned on the current FSM state. BC-FSM handles branches in the FSM in the same way as our algorithm for RSGs.

Computational Source Used for Training and Testing

In Table 5, we list the training time per epoch for our model and all baselines, and averaging time cost for each planning task when testing. All models are trained until convergence, and we list the number of epochs after which the loss stops going down.

Generative Adversarial Imitation Learning

We have also tested the Generative Adversarial Imitation Learning (GAIL)(Ho and Ermon 2016) as a baseline on the planning task on seen instructions. Since GAIL does not natually generalize to structured subgoals (FSMs), we used an LSTM to encode the task description as in the BC and IRL baselines. On CraftingWorld, GAIL achieves 19.2% planning success rate on the Comp. split, and 1.0% success rate on the Novel split. GAIL has a better performance compared to BC and BC-FSM, which we think is because it explores the environment during training. There is still a large performance gap between GAIL and RSG, mainly because GAIL does not have goal representations that helps compositional generalization.