Multi-task Hierarchical Reinforcement Learning for Compositional Tasks

by

Sungryull Sohn

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2021

Doctoral Committee:

Professor Honglak Lee, Chair
Professor Joyce Chai
Professor Satinder Singh
Professor Lei Ying

Sungryull Sohn
srsohn@umich.edu
ORCID iD: 0000-0001-7733-4293

ACKNOWLEDGMENTS

Most of all, I would like to thank God for your unlimited grace which has guided me to the best place throughout my PhD. I would also like to thank my parents, grand-parents, sisters, and brothers-in-law for supporting me through the countless prayers, advice, and love. In difficult times, you have been the greatest source of comfort and encouragement. I would like to extend my appreciation to all the people who guided and helped my academic career throughout my study. There are too many to list, but to name a few:

First of all, I would like to sincerely thank my advisor, Professor Honglak Lee, for his careful guidance and endless support throughout my studies. It was a great privilege to work with him in the works presented in this dissertation. Junhyuk Oh for guiding me to the field of reinforcement learning and providing me with essential advice for growing as a researcher. Jongwook Choi for the continuous collaboration and always being by my side for insightful and fun discussions.

My dissertation committee members, Professor Joyce Chai, Professor Satinder Singh, and Professor Lei Ying for providing valuable feedback on this dissertation.

All my amazing collaborators and labmates for their insightful discussions and warm friendship: Yuting Zhang, Seunghoon Hong, Junhyuk Oh, Ruben Villegas, Xinchen Yan, Kibok Lee, Lajanugen Logeswaran, Rui Zhang, Jongwook Choi, Yijie Guo, Yunseok Jang, Wilka Carvalho, Anthony Liu, Kimin Lee, Thomas Huang, Hyunjae Woo, Sungtae Lee, and Chris Hoang.

Mehdi Fatemi, Harm van Seijen, Jayden Ooi, Yinlam Chow, Ofir Nachuum, Minmin Chen, Ed Chi, Craig Boutilier, Aleksandra Faust, Izzeddin Gur and many others for their advice, collaborations and guidance to grow as a researcher during and after my internship at Microsoft Research and Google Brain.

Lastly, I would like to thank the University of Michigan for supporting me throughout my PhD.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

LIST OF PROGRAMS

LIST OF APPENDICES

ABSTRACT

This thesis presents the algorithms for solve multiple compositional tasks with high sample efficiency and strong generalization ability. Central to this work is the *subtask graph* which models the structure in compositional tasks into a graph form. We formulate the compositional tasks as a multi-task and meta-RL problems using the subtask graph and discuss different approaches to tackle the problem. Specifically, we present four contributions, where the common idea is to exploit the inductive bias in the hierarchical task structure for efficien learning and strong generalization.

The first part of the thesis formally introduces the subtask graph execution problem: a modeling of the compositional task as an multi-task RL problem where the agent is given a task description input in a graph form as an additional input. We present the hierarchical architecture where high-level policy determines the subtask to execute and low-level policy executes the given subtask. The high-level policy learns the modular neural network that can be dynamically assmbled according to the input task description to choose the optimal sequence of subtasks to maximize the reward. We demonstrate that the proposed method can achieve a strong zero-shot task generalization ability, and also improve the search efficiency of existing planning method when combined together.

The second part studies the more general setting where the task structure is not available to agent such that the task should be inferred from the agent's own experience; *i.e.*, few-shot reinforcement learning setting. Specifically, we combine the meta-reinforcemenet learning with an inductive logic programming (ILP) method to explicitly infer the latent task structure in terms of subtask graph from agent's trajectory. Our empirical study shows that the underlying task structure can be accurately inferred from a small amount of environment interaction without any explicit supervision on complex 3D environments with high-dimensional state and actions space.

The third contribution extends the second contribution by transfer-learning the prior over the task structure from training tasks to the unseen testing task to achieve a faster adaptation. Although the meta-policy learned the general exploration strategy over the distribution of tasks, the task structure was independently inferred from scratch for each task in the previous part. We overcome such limitation by modeling the prior of the tasks from the subtask graph inferred via ILP, and transfer-learning the learned prior when performing the inference of novel test tasks. To achieve this, we propose a novel prior sampling and posterior update method to incorporate the knowledge

learned from the seen task that is most relevant to the current task.

The fourth contribution extends the second contribution via learning a factored form of subtask graph. Specifically, the main idea is to further decompose each subtask into "entities" where we assume a certain similarity between subtasks with similar entities. We develop an algorithm that can capture the factored structure, which enables more efficient knowledge sharing between subtasks and also a stronger form of generalization to unseen subtasks with unseen entities.

The last part investigates more indirect form of inductive bias that is implemented as a constraint on the trajectory rolled out by the policy in MDP. We present a theoretical result proving that the proposed constraint preserves the optimality while reducing the policy search space. Empirically, the proposed method improves the sample effciency of the policy gradient method on a wide range of challenging sparse-reward tasks.

Overall, this work formulates the hierarchical structure in the compositional tasks and provides the evidences that such structure exists in many important problems. In addition, we present diverse principled approaches to exploit the inductive bias on the hierarchical structure in MDP in different problem settings and assumptions, and demonstrate the usefulness of such inductive bias when tackling compositional tasks.

CHAPTER 1

Introduction

The reinforcement learning (RL) is an area of machine learning that regards the agent learning to take actions in an environment [Sutton and Barto, 2018]. When the agent takes an action, the environment returns the reward and the next state to the agent. The goal of agent is to learn the policy – how to map situations (*i.e.*, state) to actinos – so as to maximize the cumulative reward. RL provides a general learning framework since the learner does not requires to be told which actions to take but can learn only from the *reward* feedback. The generality of the RL framework made it possible to formulate many challenging problems into RL problem which have achieved some successes in variaty of domains [Tesauro, 1995, Diuk et al., 2008, Riedmiller et al., 2009]. However, the main bottleneck of applying the reinforcement learning to the real-world applications was the necessity for hand-engineered features which require domain-specific knowledge.

More recently, combining reinforcement learning and deep learning opened up many new applications (*e.g.*, healthcare, robotics, etc) and achieved remarkable successes on many challenging tasks [Mnih et al., 2015, Silver et al., 2017, Vinyals et al., 2019] with high dimensional state-space where the traditional RL approaches had a difficulty of designing the feature. Deep RL used a function approximation for a policy to approximately model the mapping from state to the action using neural network. With the neural network's capacity to capture the hierarchical levels of abstractions from data, the deep RL has made many significant breakthroughs.

However, prior work mostly have been focused on a single known task where the agent can be trained for a long time. In numerous real-world scenarios, interacting with the environment is expensive or limited, and the agent is often presented with a novel task that is not seen during its training time. Thus, in this case, the agent should be able to solve multiple tasks with varying sources of reward and supervision. Recent work in multi-task RL has attempted to address this; however, they focused on the setting where the structure of task are *explicitly* described with natural language instructions [Oh et al., 2017, Andreas et al., 2017, Yu et al., 2017, Chaplot et al., 2018], programs [Denil et al., 2017], or graph structures [Sohn et al., 2018]. However, such task descriptions may not readily be available. A more flexible solution is to have the agents infer the task by interacting with the environment. Recent work in Meta RL [Hochreiter et al., 2001,

Duan et al., 2016, Wang et al., 2016, Finn et al., 2017] (especially in few-shot learning settings) has attempted to have the agents implicitly infer tasks and quickly adapt to them. However, they have focused on relatively simple tasks with a single goal (*e.g.*, multi-armed bandit, locomotion, navigation, *etc.*).

We argue that real-world tasks often have a compositional task structure and multiple goals, which require long horizon planning or reasoning ability [Erol, 1996, Xu et al., 2017, Ghazanfari and Taylor, 2017, Sohn et al., 2018]. Take, for example, the task of making a breakfast. A meal can be served with different dishes and drinks (*e.g.*, *boiled egg* and *coffee*), where each could be considered as a subtask. These can then be further decomposed into smaller substask until some base *subtask* (*e.g.*, *pickup egg*) is reached. Each subtask can provide the agent with reward; if only few subtasks provide reward, this is considered a *sparse reward* problem. In addition, the subtasks may have complex dependencies in terms of the *precondition*. For example, a bread should be sliced before toasted, or an omelette and an egg sandwich cannot be made together if there is only one egg left. Due to such complex dependencies as well as different rewards and costs, it is often cumbersome for human users to manually provide the optimal sequence of subtasks (e.g., "fry an egg and toast a bread"). Instead, the agent should learn to act in the environment by figuring out the optimal sequence of subtasks that gives the maximum reward within a time budget just from properties and dependencies of subtasks or even infer such dependencies by trying out the subtasks in different order. Thus, in real-world scenarios, the agent is required to solve multiple, potentially unseen, and compositional tasks.

The main goal of this thesis is to develop an intelligent agent that can perform many compositional tasks with high sample efficiency and strong generalization ability. Specifically, we tackle the compositional tasks by exploiting the inductive bias on the task structure.

The first part of the thesis discusses how to formulate such compositional tasks as an multi-task RL problem where the agent is given a task description input in a graph form as an additional input. Specifically, we present the subtask graph framework that is built upon the options framework Sutton [1998], and learns the modular neural network that can be dynamically assmbled according to the task description to choose the optimal subtask to execute that can maximize its reward. We demonstrate that the proposed method can achieve a strong zero-shot task generalization ability. In addition, we present how such neural network can be combined with existing planning method to improve the search efficiency and further improve the performance.

The second part studies the compositional tasks in a more general setting where the task structure is not available to agent; *i.e.*, few-shot reinforcement learning setting. Specifically, we combine the meta-reinforcemenet learning with an inductive logic programming (ILP) method to explicitly infer the latent task structure in terms of subtask graph from agent's trajectory. We demonstrate that 1) the meta-policy learns to explore the subtask space such that the task can be accurately

inferred from the collected experience of agent and 2) the highly efficient abstraction captured by ILP algorithm enables a faster adaptation of the proposed approach. We apply the proposed method to more complex 3D environments with high-dimensional state and actions space, and show that the underlying task structure can be accurately inferred from a small amount of environment interaction without any explicit supervision.

In the third part of the thesis, we extend the previous meta-RL approaches by transfer-learning the prior over the task structure from training tasks to the unseen testing task to achieve a faster adaptation. Although the meta-policy learned the general exploration strategy over the distribution of tasks, the task structure was independently inferred from scratch for each task in the previous part. We overcome such limitation by modeling the prior of the tasks from the subtask graph inferred via ILP, and transfer-learning the learned prior when performing the inference of novel test tasks. To achieve this, we propose a novel prior sampling and posterior update method to incorporate the knowledge learned from the seen task that is most relevant to the current task.

Lastly, we discuss how the inductive bias on the task structure can be indirectly implemented as a constraint on the trajectory space rolled out by the policy in MDP without making any explicit assumptions. The previous three parts were assuming an explicit compositional structure in MDP in terms of subtasks and the precondition dependency between them. In this part, we relax the assumption and consider a sparse-reward task where there exists a small set of rewarding states (*i.e.*, multi-goal setting). Specifically, our intuition is to view the sparse-reward task as a graph-MDP where each of the rewarding states corresponds to a node in the MDP graph and the edge between a pair of node is the shortest-path path between the corresponding rewarding states. Then, we prove that the optimal policy of the graph-MDP is also an optimal policy of the original MDP. From this theoretical result, we present a novel constraint on MDP that improves the sample efficiency of any model-free RL method via reducing the policy search space while preserving the optimality. We demonstrate the effectiveness of the proposed method on a wide range of challenging sparse-reward tasks and discuss how the proposed inductive bias affects the policy learning.

CHAPTER 2

Background

2.1 Markov Decision Process

A task is defined as an Markov decision process (MDP) tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \rho, \gamma)$, where $\mathcal{S}$ is a state set, $\mathcal{A}$ is an action set, $\mathcal{P}$ is a transition probability, $\mathcal{R}$ is a reward function, ρ is an initial state distribution, and $\gamma \in [0, 1)$ is a discount factor. For each state s, the value of a policy π is denoted by $V^\pi(s) = \mathbb{E}^\pi[\sum_t \gamma^t r_t \mid s_0 = s]$. Then, the goal of reinforcement learning is to find the optimal policy π^* that maximizes the expected return:

$$\pi^* = \arg\max_\pi \mathbb{E}^\pi_{s\sim\rho}\Big[\sum_t \gamma^t r_t \mid s_0 = s\Big] \tag{2.1}$$

$$= \arg\max_\pi \mathbb{E}_{s\sim\rho}\left[V^\pi(s)\right]. \tag{2.2}$$

2.2 Factored Markov Decision Processes

Factored MDP (FMDP) [Boutilier et al., 1995, Schuurmans and Patrascu, 2002] is an MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where the state space $\mathcal{S}$ can be decomposed into multiple factors $\mathcal{S}_1, \cdots, \mathcal{S}_d$ such that the transition dynamics $\mathcal{P}$ and reward function $\mathcal{R}$ can be factored. The transition dynamics is factored as $p(\mathbf{s}'|\mathbf{s}, a) = \prod_i p(s'_i|\mathbf{s}, a)$. The reward function is factored as $R(\mathbf{s}, a) = \sum_i R_i(\mathbf{s}, a)$, where $R_i(\mathbf{s}, a)$ is a local reward function. The main benefit of FMDP is that it allows us to model many compositional tasks in a principled way with a compact representation such as dynamic Bayesian network [Dean and Kanazawa, 1989, Boutilier et al., 1995]. In Section 3.3, we present our formulation of the compositional tasks as an MDP paramterized by *subtask graph*, which is a family of the factored MDP.

2.3 Multi-task Reinforcement Learning

In multi-task RL, we consider an agent presented with a task drawn from some distribution as in Andreas et al. [2017], Da Silva et al. [2012]. Each task is defined by an MDP $\mathcal{M}_G = (\mathcal{S}, \mathcal{A}, \mathcal{P}_G, \mathcal{R}_G)$ parameterized by a task parameter G with a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, task-specific transition dynamics $\mathcal{P}_G$, and task-specific reward function $\mathcal{R}_G$. We assume that the task parameter $G \in \mathcal{G}$ is *available* to agent and is drawn from a distribution $P(G)$ where $\mathcal{G}$ is a set of all possible task parameters. The goal of multi-task RL is to maximize the expected reward over the whole distribution of MDPs:

$$\pi^* = \arg\max_{\pi} \int P(G) \mathbb{E}_{s \sim \pi(\cdot;G), \mathcal{M}_G} \left[\sum_{t=0}^{T} \gamma^t \mathcal{R}_G(s_t) \right] dG, \tag{2.3}$$

where γ is a discount factor, $\pi : \mathcal{S} \times \mathcal{G} \to \mathcal{A}$ is a multitask policy that we aim to learn, and $\mathcal{R}_G$ is the task-specific reward function. We consider a zero-shot generalization where only a subset of tasks $\mathcal{G}_{train} \subset \mathcal{G}$ is available to agent during training, and the agent is required to generalize over a set of unseen tasks $\mathcal{G}_{test} \subset \mathcal{G}$ for evaluation, where $\mathcal{G}_{test} \cap \mathcal{G}_{train} = \phi$. In Chapter 3, we tackle the compositional tasks in multi-task RL setting.

2.4 Few-shot Reinforcement Learning

Different from the multi-task RL problem, the few-shot RL setting does *not* assume any task parameter input; instead, the agent is given a short period of time to *infer* about the task via interacting with the environment. Similar to multi-task RL, a *task* is defined by an MDP $\mathcal{M}_G = (\mathcal{S}, \mathcal{A}, \mathcal{P}_G, \mathcal{R}_G)$ parameterized by a task parameter G with a shared state set $\mathcal{S}$, shared action set $\mathcal{A}$, task-specific transition dynamics $\mathcal{P}_G$, and task-specific reward function $\mathcal{R}_G$. In the K-shot RL formulation [Duan et al., 2016, Finn et al., 2017], each *trial* under a fixed task $\mathcal{M}_G$ consists of an *adaptation phase* where the agent learns a task-specific behavior for K environment steps and a *test phase* where the adapted behavior is evaluated in terms of the expected discounted return. For example, RNN-based meta-learners [Duan et al., 2016, Wang et al., 2016] adapt to a task $\mathcal{M}_G$ by updating its RNN states (or fast-parameters) ϕ_t, where the initialization and update rule of ϕ_t is parameterized by a *slow-parameter* θ: $\phi_0 = g_\theta(s_0),\ \phi_{t+1} = f_\theta(\phi_t, a_t, r_t, s_{t+1})$. Gradient-based meta-learners [Finn et al., 2017, Nichol et al., 2018] instead aim to learn a good initialization of the model so that it can adapt to a new task with few gradient update steps. In the test phase, the agent's

performance on the task $\mathcal{M}_G$ is measured in terms of the return:

$$\mathcal{R}_{\mathcal{M}_G}(\pi_{\phi_K}) = \mathbb{E}_{\pi_{\phi_K}, \mathcal{M}_G}\left[\sum_{t=1}^{H} r_t\right], \tag{2.4}$$

$$\phi_0 = g_\theta(s_0), \tag{2.5}$$

$$\phi_{t+1} = f_\theta(\phi_t, a_t, r_t, s_{t+1}) \tag{2.6}$$

where π_{ϕ_K} is the policy after K update steps of adaptation, H is the horizon of test phase, and r_t is the reward at time t in the test phase. The goal is to find an optimal parameter θ that maximizes the expected return $\mathbb{E}_G[\mathcal{R}_{\mathcal{M}_G}(\pi_{\phi_K})]$ over a given distribution of tasks $p(G)$. In Chapter 4 and Chapter 5, we tackle the compositional tasks in few-shot reinforcement learning setting.

CHAPTER 3

Multi-task Reinforcement Learning for Compositional Task with Given Task Description

This chapter introduces a new RL problem where the agent is required to generalize to a previously-unseen environment characterized by a subtask graph which describes a set of subtasks and their dependencies. Unlike existing hierarchical multitask RL approaches that explicitly describe what the agent should do at a high level, our problem only describes properties of subtasks and relationships among them, which requires the agent to perform complex reasoning to find the optimal subtask to execute. To solve this problem, we propose a *neural subtask graph solver* (NSGS) which encodes the subtask graph using a recursive neural network embedding. To overcome the difficulty of training, we propose a novel non-parametric gradient-based policy, *graph reward propagation*, to pre-train our NSGS agent and further finetune it through actor-critic method. The experimental results on two 2D visual domains show that our agent can perform complex reasoning to find a near-optimal way of executing the subtask graph and generalize well to the unseen subtask graphs. In addition, we compare our agent with a Monte-Carlo tree search (MCTS) method showing that our method is much more efficient than MCTS, and the performance of NSGS can be further improved by combining it with MCTS.

3.1 Introduction

Developing the ability to execute many different tasks depending on given task descriptions and generalize over unseen task descriptions is an important problem for building scalable reinforcement learning (RL) agents. Recently, there have been a few attempts to define and solve different forms of task descriptions such as natural language [Oh et al., 2017, Yu et al., 2017] or formal language [Denil et al., 2017, Andreas et al., 2017]. However, most of the prior works have focused on task descriptions which explicitly specify what the agent should do at a high level, which may not be readily available in real-world applications.

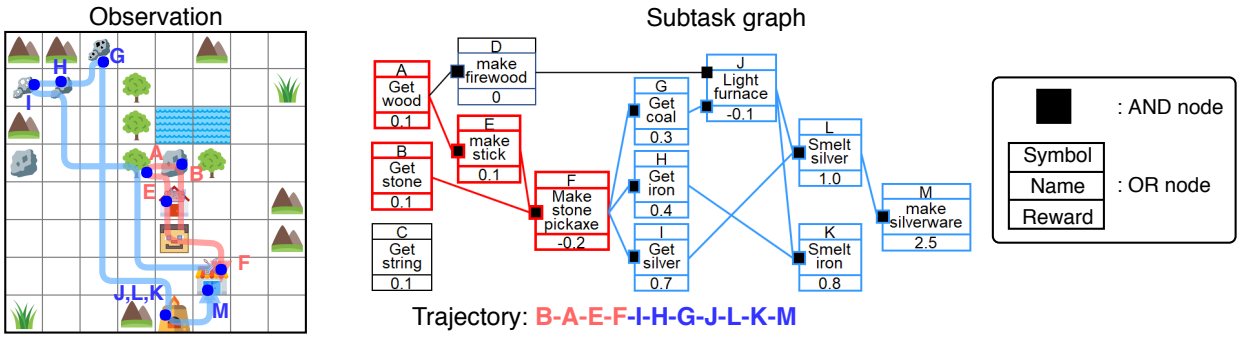


Figure 3.1: Example task and our agent's trajectory. The agent is required to execute subtasks in the optimal order to maximize the reward within a time limit. The subtask graph describes subtasks with the corresponding rewards (e.g., subtask L gives 1.0 reward) and dependencies between subtasks through AND and OR nodes. For instance, the agent should first get the firewood (D) OR coal (G) to light a furnace (J). In this example, our agent learned to execute subtask F and its preconditions (shown in red) as soon as possible, since it is a precondition of many subtasks even though it gives a negative reward. After that, the agent mines minerals that require stone pickaxe and craft items (shown in blue) to achieve a high reward.

To further motivate the problem, let's consider a scenario in which an agent needs to generalize to a complex novel task by performing a composition of subtasks where the task description and dependencies among subtasks may change depending on the situation. For example, a human user could ask a physical household robot to make a meal in an hour. A meal may be served with different combinations of dishes, each of which takes a different amount of cost (e.g., time) and gives a different amount of reward (e.g., user satisfaction) depending on the user preferences. In addition, there can be complex dependencies between subtasks. For example, a bread should be sliced before toasted, or an omelette and an egg sandwich cannot be made together if there is only one egg left. Due to such complex dependencies as well as different rewards and costs, it is often cumbersome for human users to manually provide the optimal sequence of subtasks (e.g., "fry an egg and toast a bread"). Instead, the agent should learn to act in the environment by figuring out the optimal sequence of subtasks that gives the maximum reward within a time budget just from properties and dependencies of subtasks.

The goal of this chapter is to formulate and solve such a problem, which we call *subtask graph execution*, where the agent should execute the given *subtask graph* in an optimal way as illustrated in Figure 4.1. A subtask graph consists of subtasks, corresponding rewards, and dependencies among subtasks in logical expression form where it subsumes many existing forms (e.g., sequential instructions [Oh et al., 2017]). This allows us to define many complex tasks in a principled way and train the agent to find the optimal way of executing such tasks. Moreover, we aim to solve the problem without explicit search or simulations so that our method can be more easily applicable to practical real-world scenarios, where real-time performance (i.e., fast decision-making) is required

and building the simulation model is extremely challenging.

To solve the problem, we propose a new deep RL architecture, called *neural subtask graph solver* (NSGS), which encodes a subtask graph using a recursive-reverse-recursive neural network (R3NN) [Parisotto et al., 2016] to consider the long-term effect of each subtask. Still, finding the optimal sequence of subtasks by reflecting the long-term dependencies between subtasks and the context of observation is computationally intractable. Therefore, we found that it is extremely challenging to learn a good policy when it's trained from scratch. To address the difficulty of learning, we propose to pre-train the NSGS to approximate our novel non-parametric policy called *graph reward propagation policy*. The key idea of the graph reward propagation policy is to construct a differentiable representation of the subtask graph such that taking a gradient over the reward results in propagating reward information between related subtasks, which is used to find a reasonably good subtask to execute. After the pre-training, our NSGS architecture is finetuned using the actor-critic method.

The experimental results on 2D visual domains with diverse subtask graphs show that our agent implicitly performs complex reasoning by taking into account long-term subtask dependencies as well as the cost of executing each subtask from the observation, and it can successfully generalize to unseen and larger subtask graphs. Finally, we show that our method is computationally much more efficient than Monte-Carlo tree search (MCTS) algorithm, and the performance of our NSGS agent can be further improved by combining with MCTS, achieving a near-optimal performance.

Our contributions can be summarized as follows: (1) We propose a new challenging RL problem and domain with a richer and more general form of graph-based task descriptions compared to the recent works on multitask RL. (2) We propose a deep RL architecture that can execute arbitrary *unseen* subtask graphs and observations. (3) We demonstrate that our method outperforms the state-of-the-art search-based method (e.g., MCTS), which implies that our method can efficiently approximate the solution of an intractable search problem without performing any search. (4) We further show that our method can also be used to augment MCTS, which significantly improves the performance of MCTS with a much less amount of simulations.

3.2 Related Work

Programmable Agent The idea of learning to execute a given program using RL was introduced by programmable hierarchies of abstract machines (PHAMs) [Parr and Russell, 1997, Andre and Russell, 2000, 2002]. PHAMs specify a partial policy using a set of hierarchical finite state machines, and the agent learns to execute the partial program. A different way of specifying a partial policy was explored in the deep RL framework [Andreas et al., 2017]. Other approaches used a program as a form of task description rather than a partial policy in the context of multitask RL [Oh et al.,

2017, Denil et al., 2017]. Our work also aims to build a programmable agent in that we train the agent to execute a given task. However, most of the prior work assumes that the program specifies what to do, and the agent just needs to learn how to do it. In contrast, our work explores a new form of program, called *subtask graph* (see Figure 4.1), which describes properties of subtasks and dependencies between them, and the agent is required to figure out what to do as well as how to do it.

Hierarchical Reinforcement Learning Many hierarchical RL approaches have been proposed to solve complex decision problems via multiple levels of temporal abstractions [Sutton et al., 1999b, Dietterich, 2000, Precup, 2000, Ghavamzadeh and Mahadevan, 2003, Konidaris and Barto, 2007]. Our work builds upon the prior work in that a high-level controller focuses on finding the optimal subtask, while a low-level controller focuses on executing the given subtask. In this work, we focus on how to train the high-level controller for generalizing to novel complex dependencies between subtasks.

Classical Search-Based Planning One of the most closely related problems is the planning problem considered in hierarchical task network (HTN) approaches [Sacerdoti, 1975, Erol, 1996, Erol et al., 1994, Nau et al., 1999, Castillo et al., 2005] in that HTNs also aim to find the optimal way to execute tasks given subtask dependencies. However, they aim to execute a single goal task, while the goal of our problem is to maximize the cumulative reward in RL context. Thus, the agent in our problem not only needs to consider dependencies among subtasks but also needs to infer the cost from the observation and deal with stochasticity of the environment. These additional challenges make it difficult to apply such classical planning methods to solve our problem.

Motion Planning Another related problem to our subtask graph execution problem is motion planning (MP) problem [Asano et al., 1985, Canny, 1985, 1987, Faverjon and Tournassoud, 1987, Keil and Sack, 1985]. MP problem is often mapped to a graph, and reduced to a graph search problem. However, different from our problem, the MP approaches aim to find an optimal path to the goal in the graph while avoiding obstacles similar to HTN approaches.

3.3 Problem Definition

The *subtask graph execution* problem is a multitask RL problem with a specific form of task parameter G called *subtask graph*. Figure 4.1 illustrates an example subtask graph and environment. The task of our problem is to execute given N subtasks in an optimal order to maximize reward within a time budget, where there are complex dependencies between subtasks defined by the

subtask graph. We assume that the agent has learned a set of *options* ($\mathcal{O}$) Precup [2000], Stolle and Precup [2002], Sutton et al. [1999b] that performs subtasks by executing one or more primitive actions.

Subtask Graph and Environment We define the terminologies as follows:

- **Precondition**: A *precondition* of subtask is defined as a logical expression of subtasks in sum-of-products (SoP) form where multiple `AND` terms are combined with an `OR` term (e.g., the precondition of subtask J in Figure 4.1 is `OR(AND(D), AND(G))`.
- **Eligibility vector**: $\mathbf{e}_t = [e_t^1, \ldots, e_t^N]$ where $e_t^i = 1$ if subtask i is *eligible* (i.e., the precondition of subtask is satisfied and it has never been executed by the agent) at time t, and 0 otherwise.
- **Completion vector**: $\mathbf{x}_t = [x_t^1, \ldots, x_t^N]$ where $x_t^i = 1$ if subtask i has been executed by the agent while it is eligible, and 0 otherwise.
- **Subtask reward vector**: $\mathbf{r} = [r^1, \ldots, r^N]$ specifies the reward for executing each subtask.
- **Reward**: $r_t = r^i$ if the agent executes the subtask i while it is eligible, and $r_t = 0$ otherwise.
- **Time budget**: $step_t \in \mathbb{R}$ is the remaining time-steps until episode termination.
- **Observation**: $\mathbf{obs}_t \in \mathbb{R}^{H \times W \times C}$ is a visual observation at time t as illustrated in Figure 4.1.

To summarize, a subtask graph G defines N subtasks with corresponding rewards $\mathbf{r}$ and the preconditions. The state input at time t consists of $\mathbf{s}_t = \{\mathbf{obs}_t, \mathbf{x}_t, \mathbf{e}_t, step_t\}$. The goal is to find a policy $\pi : \mathbf{s}_t, G \mapsto \mathbf{o}_t$ which maps the given context of the environment to an *option* ($\mathbf{o}_t \in \mathcal{O}$).

Challenges Our problem is challenging due to the following aspects:

- **Generalization**: Only a subset of subtask graphs ($\mathcal{G}_{train}$) is available during training, but the agent is required to execute previously unseen and larger subtask graphs ($\mathcal{G}_{test}$).
- **Complex reasoning**: The agent needs to infer the long-term effect of executing individual subtasks in terms of reward and cost (e.g., time) and find the optimal sequence of subtasks to execute without any explicit supervision or simulation-based search. We note that it may not be easy even for humans to find the solution without explicit search due to the exponentially large solution space.
- **Stochasticity**: The outcome of subtask execution is stochastic in our setting (for example, some objects are randomly moving). Therefore, the agent needs to consider the expected outcome when deciding which subtask to execute.

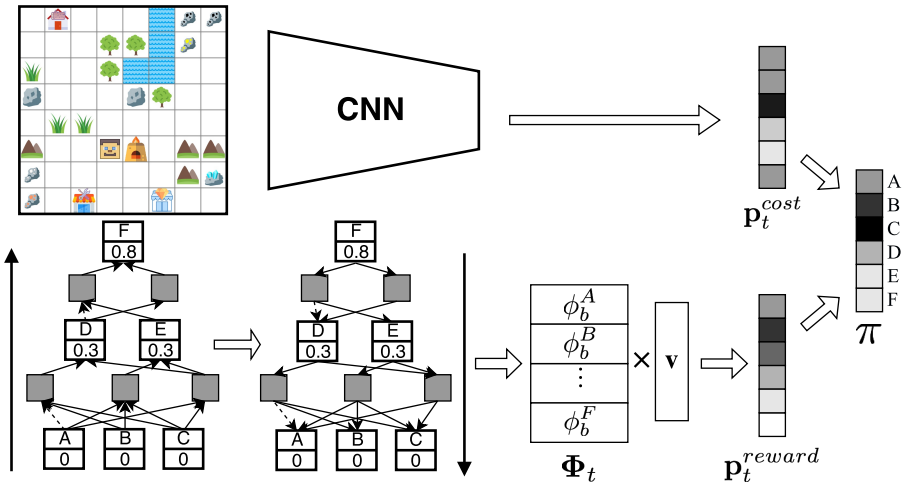


Figure 3.2: Neural subtask graph solver architecture. The task module encodes subtask graph through a bottom-up and top-down process, and outputs the reward score $\mathbf{p}_t^{reward}$. The observation module encodes observation using CNN and outputs the cost score $\mathbf{p}_t^{cost}$. The final policy is a softmax policy over the sum of two scores.

3.4 Method

Our *neural subtask graph solver* (NSGS) is a neural network which consists of a *task module* and an *observation module* as shown in Figure 3.2. The task module encodes the precondition of each subtask via bottom-up process and propagates the information about future subtasks and rewards to preceding subtasks (i.e., pre-conditions) via the top-down process. The observation module learns the correspondence between a subtask and its target object, and the relation between the locations of objects in the observation and the time cost. However, due to the aforementioned challenge (i.e., *complex reasoning*), learning to execute the subtask graph only from the reward is extremely challenging. To facilitate the learning, we propose *graph reward propagation policy* (GRProp), a non-parametric policy that propagates the reward information between related subtasks to model their dependencies. Since our GRProp acts as a good initial policy, we train the NSGS to approximate the GRProp policy through policy distillation [Rusu et al., 2015, Parisotto et al., 2015], and finetune it through actor-critic method with generalized advantage estimation (GAE) [Schulman et al., 2016] to maximize the reward. Section 3.4.1 describes the NSGS architecture, and Section 3.4.2 describes how to construct the GRProp policy.

3.4.1 Neural Subtask Graph Solver

Task Module Given a subtask graph G, the remaining time steps $step_t \in \mathbb{R}$, an eligibility vector $\mathbf{e}_t$ and a completion vector $\mathbf{x}_t$, we compute a context embedding using recursive-reverse-recursive

neural network (R3NN) Parisotto et al. [2016] as follows:

$$\phi^i_{bot,o} = b_{\theta_o}\left(x^i_t, e^i_t, step_t, \sum_{j\in Child_i}\phi^j_{bot,a}\right), \qquad \phi^j_{bot,a} = b_{\theta_a}\left(\sum_{k\in Child_j}\left[\phi^k_{bot,o}, w^{j,k}_+\right]\right), \tag{3.1}$$

$$\phi^i_{top,o} = t_{\theta_o}\left(\phi^i_{bot,o}, r^i, \sum_{j\in Par_i}\left[\phi^j_{top,a}, w^{i,j}_+\right]\right), \qquad \phi^j_{top,a} = t_{\theta_a}\left(\phi^j_{bot,a}, \sum_{k\in Par_j}\phi^k_{top,o}\right), \tag{3.2}$$

where $[\cdot]$ is a concatenation operator, b_θ, t_θ are the bottom-up and top-down encoding function, $\phi^i_{bot,a}$, $\phi^i_{top,a}$ are the bottom-up and top-down embedding of i-th `AND` node respectively, and $\phi^i_{bot,o}$, $\phi^i_{top,o}$ are the bottom-up and top-down embedding of i-th `OR` node respectively (see Appendix for the detail). The $w^{i,j}_+$, $Child_i$, and $Parent_i$ specifies the connections in the subtask graph G. Specifically, $w^{i,j}_+ = 1$ if j-th `OR` node and i-th `AND` node are connected without `NOT` operation, -1 if there is `NOT` connection and 0 if not connected, and $Child_i, Parent_i$ represent a set of i-th node's children and parents respectively. The embeddings are transformed to reward scores via: $\mathbf{p}^{reward}_t = \mathbf{\Phi}^\top_{top}\mathbf{v}$, where $\mathbf{\Phi}_{top} = [\phi^1_{top,o}, \ldots, \phi^N_{top,o}] \in \mathbb{R}^{E\times N}$, E is the dimension of the top-down embedding of `OR` node, and $\mathbf{v} \in \mathbb{R}^E$ is a weight vector for reward scoring.

Observation Module The observation module encodes the input observation $\mathbf{s}_t$ using a convolutional neural network (CNN) and outputs a cost score:

$$\mathbf{p}^{cost}_t = \text{CNN}(\mathbf{s}_t, step_t). \tag{3.3}$$

where $step_t$ is the number of remaining time steps. An ideal observation module would learn to estimate high score for a subtask if the target object is close to the agent because it would require less cost (i.e., time). Also, if the expected number of step required to execute a subtask is larger than the remaining step, ideal agent would assign low score. The NSGS policy is a softmax policy:

$$\pi(\mathbf{o}_t|\mathbf{s}_t, \mathbf{G}, \mathbf{x}_t, \mathbf{e}_t, step_t) = \text{Softmax}(\mathbf{p}^{reward}_t + \mathbf{p}^{cost}_t), \tag{3.4}$$

which adds reward scores and cost scores.

3.4.2 Graph Reward Propagation Policy: Pre-training Neural Subtask Graph Solver

Intuitively, the graph reward propagation policy is designed to put high probabilities over subtasks that are likely to maximize the sum of *modified and smoothed* reward $\widetilde{U}_t$ at time t, which will be

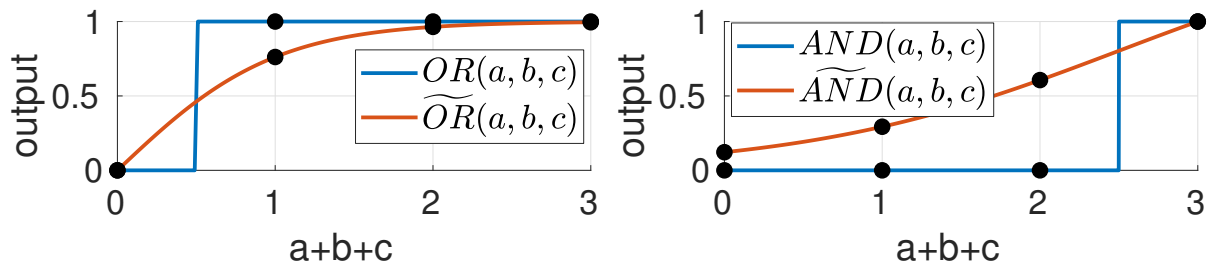


Figure 3.3: Visualization of OR, $\widetilde{\text{OR}}$, AND, and $\widetilde{\text{AND}}$ operations with three inputs (a,b,c). These smoothed functions are defined to handle arbitrary number of operands (see Appendix).

defined in Eq. 3.11. Let $\mathbf{x}_t$ be a completion vector and $\mathbf{r}$ be a subtask reward vector (see Section 3.3 for definitions). Then, the sum of reward until time-step t is given as:

$$U_t = \mathbf{r}^T \mathbf{x}_t. \tag{3.5}$$

We first modify the reward formulation such that it gives a half of subtask reward for satisfying the preconditions and the rest for executing the subtask to encourage the agent to satisfy the precondition of a subtask with a large reward:

$$\hat{U}_t = \mathbf{r}^T (\mathbf{x}_t + \mathbf{e}_t)/2. \tag{3.6}$$

Let y^j_{AND} be the output of j-th AND node. The eligibility vector $\mathbf{e}_t$ can be computed from the subtask graph G and $\mathbf{x}_t$ as follows:

$$e^i_t = \underset{j \in Child_i}{\text{OR}} \left(y^j_{AND} \right), \tag{3.7}$$

$$y^j_{AND} = \underset{k \in Child_j}{\text{AND}} \left(\hat{x}^{j,k}_t \right), \tag{3.8}$$

$$\hat{x}^{j,k}_t = x^k_t w^{j,k} + (1 - x^k_t)(1 - w^{j,k}), \tag{3.9}$$

where $w^{j,k} = 0$ if there is a NOT connection between j-th node and k-th node, otherwise $w^{j,k} = 1$. Intuitively, $\hat{x}^{j,k}_t = 1$ when k-th node does not violate the precondition of j-th node. Note that $\tilde{U}_t$ is not differentiable with respect to $\mathbf{x}_t$ because AND$(\cdot)$ and OR$(\cdot)$ are not differentiable. To derive our graph reward propagation policy, we propose to substitute AND$(\cdot)$ and OR$(\cdot)$ functions with "smoothed" functions $\widetilde{\text{AND}}$ and $\widetilde{\text{OR}}$ as follows:

$$\widetilde{e}^i_t = \underset{j \in Child_i}{\widetilde{\text{OR}}} \left(\widetilde{y}^j_{AND} \right), \qquad \widetilde{y}^j_{AND} = \underset{k \in Child_j}{\widetilde{\text{AND}}} \left(\hat{x}^{j,k}_t \right), \tag{3.10}$$

where $\widetilde{\text{AND}}$ and $\widetilde{\text{OR}}$ were implemented as scaled sigmoid and tanh functions as illustrated by Figure 3.3 (see Appendix for details). With the smoothed operations, the sum of smoothed and modified reward is given as:

$$\widetilde{U}_t = \mathbf{r}^T(\mathbf{x}_t + \widetilde{\mathbf{e}}_t)/2. \tag{3.11}$$

Finally, the graph reward propagation policy is a softmax policy,

$$\pi(\mathbf{o}_t|G, \mathbf{x}_t) = \text{Softmax}\left(\nabla_{\mathbf{x}_t}\widetilde{U}_t\right) = \text{Softmax}\left(\frac{1}{2}\mathbf{r}^T + \frac{1}{2}\mathbf{r}^T\nabla_{\mathbf{x}_t}\widetilde{\mathbf{e}}_t\right), \tag{3.12}$$

that is the softmax of the gradient of $\widetilde{U}_t$ with respect to $\mathbf{x}_t$.

3.4.3 Policy Optimization

The NSGS is first trained through policy distillation by minimizing the KL divergence between NSGS and teacher policy (GRProp) as follows:

$$\nabla_\theta \mathcal{L}_1 = \mathbb{E}_{G\sim\mathcal{G}_{train}}\left[\mathbb{E}_{s\sim\pi_\theta^G}\left[\nabla_\theta D_{KL}\left(\pi_T^G||\pi_\theta^G\right)\right]\right], \tag{3.13}$$

where θ is the parameter of NSGS, π_θ^G is the simplified notation of NSGS policy with subtask graph G, π_T^G is the simplified notation of teacher (GRProp) policy with subtask graph G, D_{KL} is KL divergence, and $\mathcal{G}_{train}$ is the training set of subtask graphs. After policy distillation, we finetune NSGS agent in an end-to-end manner using actor-critic method with GAE [Schulman et al., 2016] as follows:

$$\nabla_\theta \mathcal{L}_2 = \mathbb{E}_{G\sim\mathcal{G}_{train}}\left[\mathbb{E}_{s\sim\pi_\theta^G}\left[-\nabla_\theta \log \pi_\theta^G \sum_{l=0}^{\infty}\left(\prod_{n=0}^{l-1}(\gamma\lambda)^{k_n}\right)\delta_{t+l}\right]\right], \tag{3.14}$$

$$\delta_t = r_t + \gamma^{k_t}V_{\theta'}^\pi(\mathbf{s}_{t+1}, G) - V_{\theta'}^\pi(\mathbf{s}_t, G), \tag{3.15}$$

where k_t is the duration of option $\mathbf{o}_t$, γ is a discount factor, $\lambda \in [0, 1]$ is a weight for balancing between bias and variance of the advantage estimation, and $V_{\theta'}^\pi$ is the critic network parameterized by θ'. During training, we update the critic network to minimize $\mathbb{E}\left[(R_t - V_{\theta'}^\pi(\mathbf{s}_t, G))^2\right]$, where R_t is the discounted cumulative reward at time t. The complete procedure for training our NSGS agent is summarized in Algorithm 6.1. We used η_d=1e-4, η_c=3e-6 for distillation and η_{ac}=1e-6, η_c=3e-7 for fine-tuning in the experiment.

for iteration n **do**
  Sample $G \sim \mathcal{G}_{train}$
  $\mathcal{D} = \{(\mathbf{s}_t, \mathbf{o}_t, r_t, R_t, step_t), \ldots\} \sim \pi_\theta^G$ ▷ do rollout
  $\theta' \leftarrow \theta' + \eta_c \sum_{\mathcal{D}} \left(\nabla_{\theta'} V_{\theta'}^{\pi}(\mathbf{s}_t, G)\right) \left(R_t - V_{\theta'}^{\pi}(\mathbf{s}_t, G)\right)$ ▷ update critic
  if distillation **then**
    $\theta \leftarrow \theta + \eta_d \sum_{\mathcal{D}} \nabla_\theta D_{KL}\left(\pi_T^G || \pi_\theta^G\right)$ ▷ update policy
  else if fine-tuning **then**
    Compute δ_t from Eq. 3.15 for all t
    $\theta \leftarrow \theta + \eta_{ac} \sum_{\mathcal{D}} \nabla_\theta \log \pi_\theta^G \sum_{l=0}^{\infty} \left(\prod_{n=0}^{l-1} (\gamma\lambda)^{k_n}\right) \delta_{t+l}$ ▷ update policy
  end if
end for

Program 3.1: Policy optimization

3.5 Experiment

In the experiment, we investigated the following research questions: 1) Does GRProp outperform other heuristic baselines (e.g., greedy policy, etc.)? 2) Can NSGS deal with complex subtask dependencies, delayed reward, and the stochasticity of the environment? 3) Can NSGS generalize to unseen subtask graphs? 4) How does NSGS perform compared to MCTS? 5) Can NSGS be used to improve MCTS?

3.5.1 Environment

We evaluated the performance of our agents on two domains: **Mining** and **Playground** that are developed based on MazeBase [Sukhbaatar et al., 2015][1]. We used a pre-trained subtask executer for each domain. The episode length (time budget) was randomly set for each episode in a range such that GRProp agent executes $60\% - 80\%$ of subtasks on average. The subtasks in the higher layer in subtask graph are designed to give larger reward (see Appendix for details).

Mining domain is inspired by Minecraft (see Figures 4.1 and 3.5). The agent may pickup raw materials in the world, and use it to craft different items on different craft stations. There are two forms of preconditions: 1) an item may be an ingredient for building other items (e.g., stick and stone are ingredients of stone pickaxe), and 2) some tools are required to pick up some objects (e.g., agent need stone pickaxe to mine iron ore). The agent can use the item multiple times after picking it once. The set of subtasks and preconditions are hand-coded based on the crafting recipes in Minecraft, and used as a template to generate 640 random subtask graphs. We used 200 for training and 440 for testing.

Playground is a more flexible and challenging domain (see Figure 3.6). The subtask graph

[1]The code is available on https://github.com/srsohn/subtask-graph-execution

in Playground was randomly generated, hence its precondition can be any logical expression and the reward may be delayed. Some of the objects randomly move, which makes the environment stochastic. The agent was trained on small subtask graphs, while evaluated on much larger subtask graphs (See Table 3.1). The set of subtasks is $\mathcal{O} = \mathcal{A}_{int} \times \mathcal{X}$, where $\mathcal{A}_{int}$ is a set of primitive actions to interact with objects, and $\mathcal{X}$ is a set of all types of interactive objects in the domain. We randomly generated 500 graphs for training and 2,000 graphs for testing. Note that the task in playground domain subsumes many other hierarchical RL domains such as Taxi [Bloch, 2009], Minecraft [Oh et al., 2017] and XWORLD [Yu et al., 2017]. In addition, we added the following components into subtask graphs to make the task more challenging:

- Distractor subtask: A subtask with only `NOT` connection to parent nodes in the subtask graph. Executing this subtask may give an immediate reward, but it may make other subtasks ineligible.
- Delayed reward: Agent receives no reward from subtasks in the lower layers, but it should execute some of them to make higher-level subtasks eligible (see Appendix for fully-delayed reward case).

3.5.2 Agents

We evaluated the following policies:

- **Random** policy executes any eligible subtask.
- **Greedy** policy executes the eligible subtask with the largest reward.
- **Optimal** policy is computed from exhaustive search on *eligible* subtasks.
- **GRProp** (Ours) is graph reward propagation policy.
- **NSGS** (Ours) is distilled from GRProp policy and finetuned with actor-critic.
- **Independent** is an LSTM-based baseline trained on each subtask graph independently, similar to Independent model in Andreas et al. [2017]. It takes the same set of input as NSGS except the subtask graph.

To our best knowledge, existing work on hierarchical RL cannot directly address our problem with a subtask graph input. Instead, we evaluated an instance of hierarchical RL method (**Independent** agent) in **adaptation** setting, as discussed in Section 3.5.3.

3.5.3 Quantitative Result

Training Performance The learning curves of NSGS and performance of other agents are shown in Figure 4.4. Our GRProp policy significantly outperforms the Greedy policy. This implies that the

Subtask Graph Setting					
	Playground				Mining
Task	**D1**	**D2**	**D3**	**D4**	**Eval**
Depth	4	4	5	6	4-10
Subtask	13	15	16	16	10-26

Zero-Shot Performance					
	Playground				Mining
Task	**D1**	**D2**	**D3**	**D4**	**Eval**
NSGS (Ours)	**.820**	**.785**	**.715**	**.527**	**8.19**
GRProp (Ours)	.721	.682	.623	.424	6.16
Greedy	.164	.144	.178	.228	3.39
Random	0	0	0	0	2.79
Adaptation Performance					
	Playground				Mining
Task	**D1**	**D2**	**D3**	**D4**	**Eval**
NSGS (Ours)	**.828**	**.797**	**.733**	**.552**	**8.58**
Independent	.346	.296	.193	.188	3.89

Table 3.1: Generalization performance on unseen and larger subtask graphs. (Playground) The subtask graphs in **D1** have the same graph structure as training set, but the graph was unseen. The subtask graphs in **D2**, **D3**, and **D4** have (unseen) larger graph structures. (Mining) The subtask graphs in **Eval** are unseen during training. NSGS outperforms other compared agents on all the task and domain.

proposed idea of back-propagating the reward gradient captures long-term dependencies among subtasks to some extent. We also found that NSGS further improves the performance through fine-tuning with actor-critic method. We hypothesize that NSGS learned to estimate the expected costs of executing subtasks from the observations and consider them along with subtask graphs.

Generalization Performance We considered two different types of generalization: a **zero-shot** setting where agent must immediately achieve good performance on unseen subtask graphs without learning, and an **adaptation** setting where agent can learn about task through the interaction with environment. Note that Independent agent was evaluated in adaptation setting only since it has no ability to generalize as it does not take subtask graph as input. Particularly, we tested agents on larger subtask graphs by varying the number of layers of the subtask graphs from four to six with a larger number of subtasks on Playground domain. Table 3.1 summarizes the results in terms of normalized reward $\bar{R} = (R - R_{min})/(R_{max} - R_{min})$ where R_{min} and R_{max} correspond to the average reward of the Random and the Optimal policy respectively. Due to large number of subtasks ($>$16) in Mining domain, the Optimal policy was intractable to be evaluated. Instead, we reported the un-normalized mean reward. Though the performance degrades as the subtask graph becomes larger as expected,

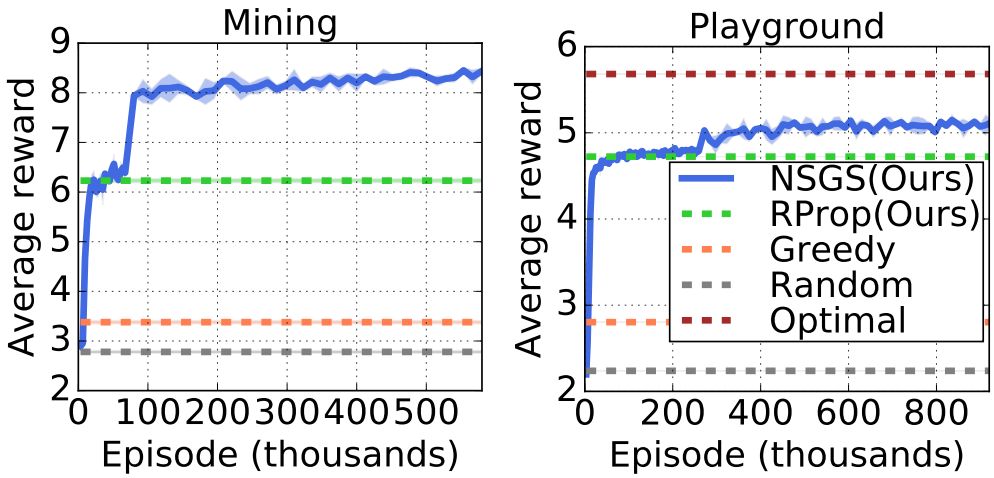


Figure 3.4: Learning curves on Mining and Playground domain. NSGS is distilled from GRProp on 77K and 256K episodes, respectively, and finetuned after that.

NSGS generalizes well to larger subtask graphs and consistently outperforms all the other agents on Playground and Mining domains in zero-shot setting. In adaptation setting, NSGS performs slightly better than zero-shot setting by fine-tuning on the subtask graphs in evaluation set. Independent agent learned a policy comparable to Greedy, but performs much worse than NSGS.

3.5.4 Qualitative Result

Figure 3.5 visualizes trajectories of agents on Mining domain. Greedy policy mostly focuses on subtasks with immediate rewards (e.g., get string, make bow) that are sub-optimal in the long run. In contrast, NSGS and GRProp agents focus on executing subtask H (make stone pickaxe) in order to collect materials much faster in the long run. Compared to GRProp, NSGS learns to consider observation also and avoids subtasks with high cost (e.g., get coal).

Figure 3.6 visualizes trajectories on Playground domain. In this graph, there are distractors (e.g., D, E, and H) and the reward is delayed. In the beginning, Greedy chooses to execute distractors, since they gives positive reward while subtasks A, B, and C do not. However, GRProp observes non-zero gradient for subtasks A, B, and C that are propagated from the parent nodes. Thus, even though the reward is delayed, GRProp can figure out which subtask to execute. NSGS learns to understand long-term dependencies from GRProp, and finds shorter path by also considering the observation.

3.5.5 Combining NSGS with Monte-Carlo Tree Search

We further investigated how well our NSGS agent performs compared to conventional search-based methods and how our NSGS agent can be combined with search-based methods to further improve

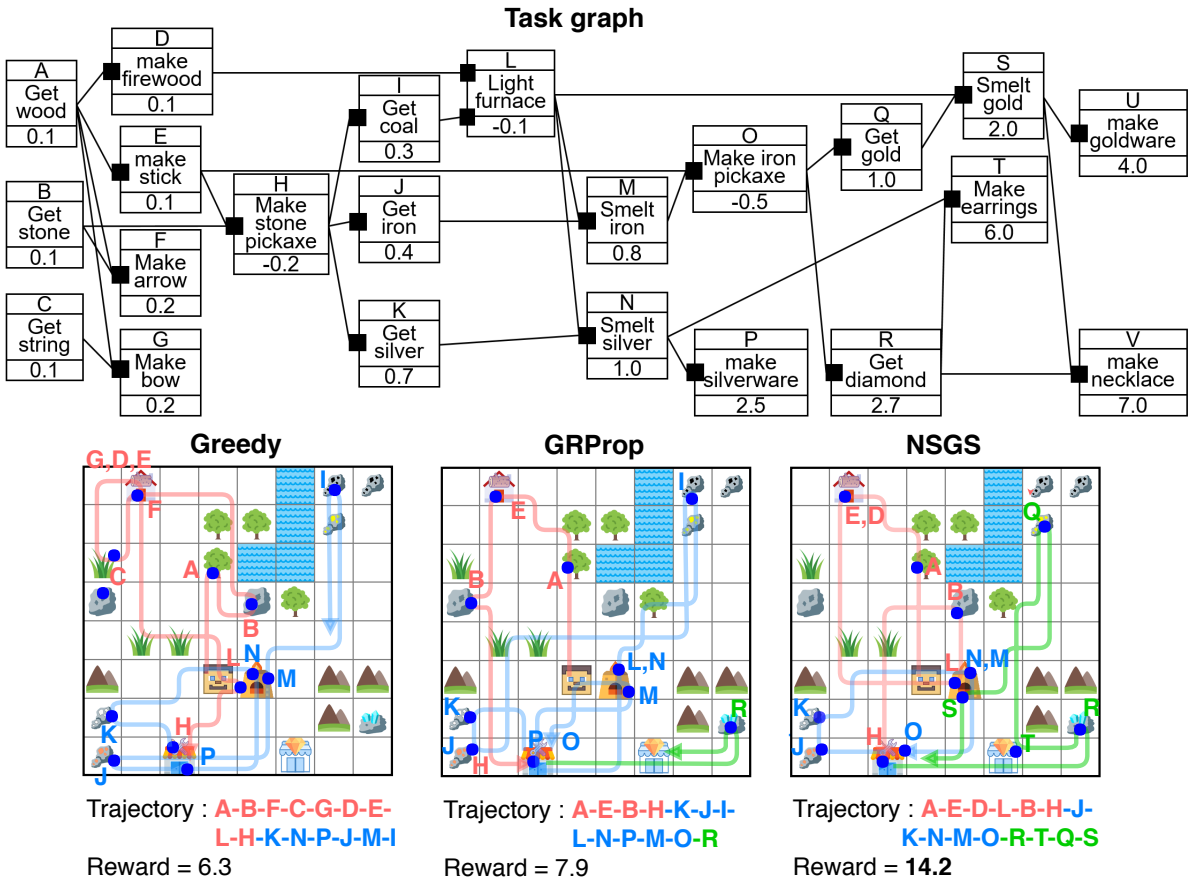


Figure 3.5: Example trajectories of Greedy, GRProp, and NSGS agents given 75 steps on Mining domain. We used different colors to indicate that agent has different types of pickaxes: red (no pickaxe), blue (stone pickaxe), and green (iron pickaxe). Greedy agent prefers subtasks C, D, F, and G to H and L since C, D, F, and G gives positive immediate reward, whereas NSGS and GRProp agents find a short path to make stone pickaxe, focusing on subtasks with higher long-term reward. Compared to GRProp, the NSGS agent can find a shorter path to make an iron pickaxe, and succeeds to execute more number of subtasks.

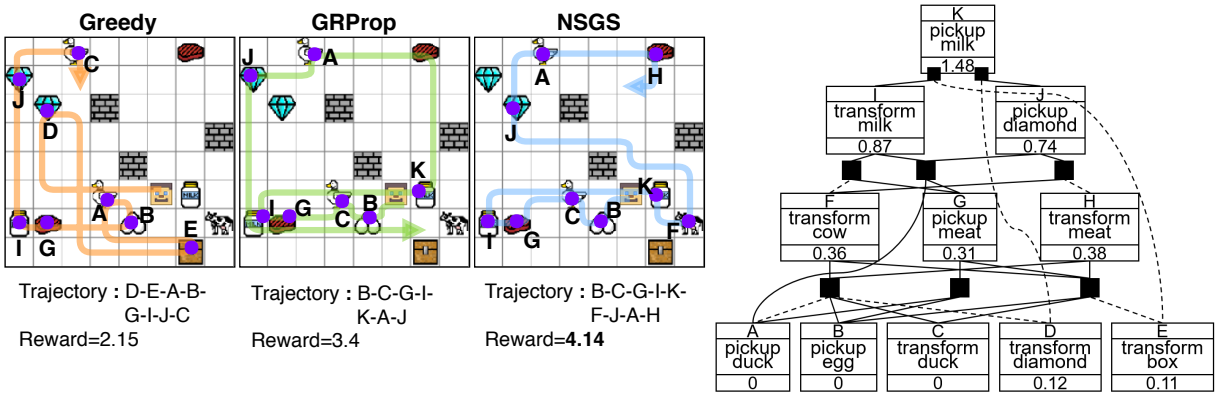


Figure 3.6: Example trajectories of Greedy, GRProp, and NSGS agents given 45 steps on Playground domain. The subtask graph includes `NOT` operation and distractor (subtask D, E, and H). We removed stochasticity in environment for the controlled experiment. Greedy agent executes the distractors since they give positive immediate rewards, which makes it impossible to execute the subtask K which gives the largest reward. GRProp and NSGS agents avoid distractors and successfully execute subtask K by satisfying its preconditions. After executing subtask K, the NSGS agent found a shorter path to execute remaining subtasks than the GRProp agent and gets larger reward.

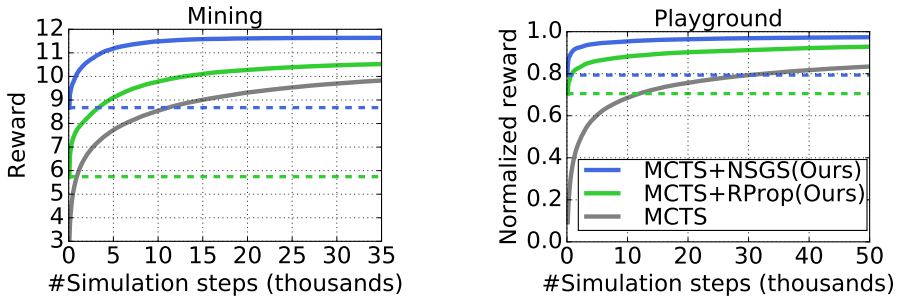


Figure 3.7: Performance of MCTS+NSGS, MCTS+GRProp and MCTS per the number of simulated steps on (Left) **Eval** of Mining domain and (Right) **D2** of Playground domain (see Table 3.1).

the performance. We implemented the following methods (see Appendix for the detail):

- MCTS: An MCTS algorithm with UCB [Auer et al., 2002] criterion for choosing actions.
- MCTS+NSGS: An MCTS algorithm combined with our NSGS agent. NSGS policy was used as a rollout policy to explore reasonably good states during tree search, which is similar to AlphaGo [Silver et al., 2016].
- MCTS+GRProp: An MCTS algorithm combined with our GRProp agent similar to MCTS+NSGS.

The results are shown in Figure 3.7. It turns out that our NSGS performs as well as MCTS method with approximately 32K simulations on Playground and 11K simulations on Mining domain, while GRProp performs as well as MCTS with approximately 11K simulations on Playground and 1K simulations on Mining domain. This indicates that our NSGS agent implicitly performs long-term reasoning that is not easily achievable by a sophisticated MCTS, even though NSGS does not use any simulation and has never seen such subtask graphs during training. More interestingly, MCTS+NSGS and MCTS+GRProp significantly outperforms MCTS, and MCTS+NSGS achieves approximately 0.97 normalized reward with 33K simulations on Playground domain. We found that the Optimal policy, which corresponds to normalized reward of 1.0, uses approximately 648M simulations on Playground domain. Thus, MCTS+NSGS performs almost as well as the Optimal policy with only 0.005% simulations compared to the Optimal policy. This result implies that NSGS can also be used to improve simulation-based planning methods by effectively reducing the search space.

3.6 Discussion

We introduced the subtask graph execution problem which is an effective and principled framework of describing complex tasks. To address the difficulty of dealing with complex subtask dependencies, we proposed a graph reward propagation policy derived from a differentiable form of subtask graph, which plays an important role in pre-training our neural subtask graph solver architecture. The empirical results showed that our agent can deal with long-term dependencies between subtasks and generalize well to unseen subtask graphs. In addition, we showed that our agent can be used to effectively reduce the search space of MCTS so that the agent can find a near-optimal solution with a small number of simulations. In this paper, we assumed that the subtask graph (e.g., subtask dependencies and rewards) is given to the agent. However, it will be very interesting future work to investigate how to extend to more challenging scenarios where the subtask graph is unknown (or partially known) and thus need to be estimated through experience.

CHAPTER 4

Meta Reinforcement Learning for Compositional Task via Task Inference

This chapter proposes and addresses a novel few-shot RL problem, where a task is characterized by a subtask graph which describes a set of subtasks and their dependencies that are unknown to the agent. The agent needs to quickly adapt to the task over few episodes during adaptation phase to maximize the return in the test phase. Instead of directly learning a meta-policy, we develop a *Meta-learner with Subtask Graph Inference* (MSGI), which infers the latent parameter of the task by interacting with the environment and maximizes the return given the latent parameter. To facilitate learning, we adopt an intrinsic reward inspired by upper confidence bound (UCB) that encourages efficient exploration. Our experiment results on two grid-world domains and StarCraft II environments show that the proposed method is able to accurately infer the latent task parameter, and to adapt more efficiently than existing meta RL and hierarchical RL methods [1].

4.1 Introduction

Recently, reinforcement learning (RL) systems have achieved super-human performance on many complex tasks [Mnih et al., 2015, Silver et al., 2016, Van Seijen et al., 2017]. However, these works mostly have been focused on a single known task where the agent can be trained for a long time (*e.g.*, Silver et al. [2016]). We argue that agent should be able to solve multiple tasks with varying sources of reward. Recent work in multi-task RL has attempted to address this; however, they focused on the setting where the structure of task are *explicitly* described with natural language instructions [Oh et al., 2017, Andreas et al., 2017, Yu et al., 2017, Chaplot et al., 2018], programs [Denil et al., 2017], or graph structures [Sohn et al., 2018]. However, such task descriptions may not readily be available. A more flexible solution is to have the agents infer the task by interacting with the environment.

[1] The demo videos and the code are available at https://bit.ly/msgi-videos and https://github.com/srsohn/msgi.

Recent work in Meta RL [Hochreiter et al., 2001, Duan et al., 2016, Wang et al., 2016, Finn et al., 2017] (especially in few-shot learning settings) has attempted to have the agents implicitly infer tasks and quickly adapt to them. However, they have focused on relatively simple tasks with a single goal (*e.g.*, multi-armed bandit, locomotion, navigation, *etc.*).

We argue that real-world tasks often have a hierarchical structure and multiple goals, which require long horizon planning or reasoning ability [Erol, 1996, Xu et al., 2017, Ghazanfari and Taylor, 2017, Sohn et al., 2018]. Take, for example, the task of making a breakfast in Figure 4.1. A meal can be served with different dishes and drinks (*e.g.*, *boiled egg* and *coffee*), where each could be considered as a subtask. These can then be further decomposed into smaller substask until some base subtask (*e.g.*, *pickup egg*) is reached. Each subtask can provide the agent with reward; if only few subtasks provide reward, this is considered a *sparse reward* problem. When the subtask dependencies are complex and reward is sparse, learning an optimal policy can require a large number of interactions with the environment. This is the problem scope we focus on in this chapter: learning to quickly infer and adapt to varying hierarchical tasks with multiple goals and complex subtask dependencies.

To this end, we formulate and tackle a new few-shot RL problem called *subtask graph inference* problem, where the task is defined as a factored MDP [Boutilier et al., 1995, Jonsson and Barto, 2006] with hierarchical structure represented by *subtask graph* [Sohn et al., 2018] where the task is not known *a priori*. The task consists of multiple subtasks, where each subtask gives reward when completed (see Figure 1). The complex dependencies between subtasks (*i.e.*, preconditions) enforce agent to execute all the required subtasks *before* it can execute a certain subtask. Intuitively, the agent can efficiently solve the task by leveraging the inductive bias of underlying task structure (Section 4.3).

Inspired by the recent works on multi-task and few-shot RL, we propose a meta reinforcement learning approach that explicitly infers the latent structure of the task (*e.g.*, subtask graph). The agent learns its adaptation policy to collect as much information about the environment as possible in order to rapidly and accurately infer the unknown task structure. After that, the agent's test policy is a contextual policy that takes the inferred subtask graph as an input and maximizes the expected return (See Figure 4.1). We leverage inductive logic programming (ILP) technique to derive an efficient task inference method based on the principle of maximum likelihood. To facilitate learning, we adopt an intrinsic reward inspired by upper confidence bound (UCB) that encourages efficient exploration. We evaluate our approach on various environments ranging from simple grid-world [Sohn et al., 2018] to StarCraft II [Vinyals et al., 2017]. In all cases, our method can accurately infer the latent subtask graph structure, and adapt more efficiently to unseen tasks than the baselines.

The contribution of this work can be summarized as follows:

- We propose a new meta-RL problem with more general and richer form of tasks compared to the

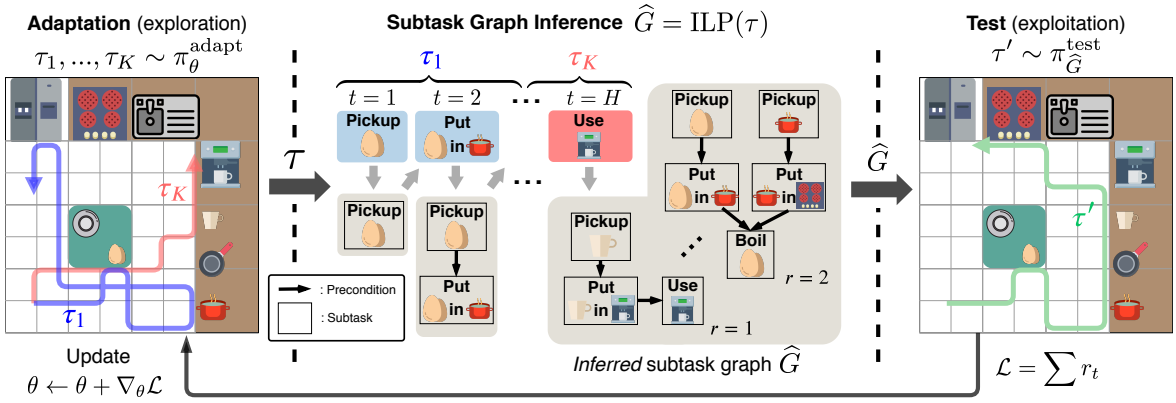


Figure 4.1: Overview of our method in the context of *prepare breakfast* task. This task can be broken down into subtasks (*e.g.*, *pickup mug*) that composes the underlying subtask graph G. (**Left**) To learn about the unknown task, the agent collects trajectories over K episodes through a parameterized *adaptation* policy $\pi_\theta^{\text{adapt}}$ that learns to explore the environment. (**Center**) With each new trajectory, the agent attempts to infer the task's underlying *ground-truth* subtask graph G with $\hat{G}$. (**Right**) A separate *test* policy $\pi_{\hat{G}}^{\text{test}}$ uses the inferred subtask graph $\hat{G}$ to produce a trajectory that attempts to maximize the agent's reward $\sum r_t$ (*e.g.*, the green trajectory that achieves the *boil egg* subtask). The more precise $\hat{G}$, the more reward the agent would receive, which implicitly improves the *adaptation* policy $\pi_\theta^{\text{adapt}}$ to better explore the environment and therefore better infer $\hat{G}$ in return.

recent meta-RL approaches.

- We propose an efficient task inference algorithm that leverages inductive logic programming, which *accurately* infers the latent subtask graph from the agent's experience data.
- We implement a deep meta-RL agent that *efficiently* infers the subtask graph for faster adaptation.
- We compare our method with other meta-RL agents on various domains, and show that our method adapts more efficiently to unseen tasks with complex subtask dependencies.

4.2 Related Work

Meta Reinforcement Learning. There are roughly two broad categories of meta-RL approaches: gradient-based meta-learners [Finn et al., 2017, Nichol et al., 2018, Gupta et al., 2018, Finn et al., 2018, Kim et al., 2018] and RNN-based meta-learners [Duan et al., 2016, Wang et al., 2016]. Gradient-based meta RL algorithms, such as MAML [Finn et al., 2017] and Reptile [Nichol et al., 2018], learn the agent's policy by taking policy gradient steps during an adaptation phase, where the meta-learner aims to learn a good initialization that enables rapid adaptation to an unseen task. RNN-based meta-RL methods [Duan et al., 2016, Wang et al., 2016] updates the hidden states of a RNN as a process of adaptation, where both of hidden state initialization and update rule are meta-learned. Other variants of adaptation models instead of RNNs such as temporal convolutions

(SNAIL) [Mishra et al., 2018] also have been explored. Our approach is closer to the second category, but different from existing works as we directly and explicitly infer the task parameter.

Logic induction. Inductive logic programming systems [Muggleton, 1991] learn a set of rules from examples. [Xu et al., 2017] These works differ from ours as they are open-loop LPI; the input data to LPI module is generated by other policy that does not care about ILP process. However, our agent learns a policy to collect data more efficiently (i.e., closed-loop ILP). There also have been efforts to combine neural networks and logic rules to deal with noisy and erroneous data and seek data efficiency, such as [Hu et al., 2016, Evans and Grefenstette, 2017, Dong et al., 2019].

Autonomous Construction of Task Structure. Task planning approaches represented the task structure using Hierarchical Task Networks (HTNs) [Tate, 1977]. HTN identifies subtasks for a given task and represent symbolic representations of their preconditions and effects, to reduce the search space of planning [Hayes and Scassellati, 2016]. They aim to execute a single goal task, often with assumptions of simpler subtask dependency structures (*e.g.*, without *NOT* dependency [Ghazanfari and Taylor, 2017, Liu et al., 2016]) such that the task structure can be constructed from the successful trajectories. In contrast, we tackle a more general and challenging setting, where each subtask gives a reward (*i.e.*, multi-goal setting) and the goal is to maximize the cumulative sum of reward within an episode. More recently, these task planning approaches were successfully applied to the few-shot visual imitation learning tasks by constructing recursive programs [Xu et al., 2017] or graph [Huang et al., 2018]. Contrary to them, we employ an *active* policy that seeks for experience useful in discovering the task structure in unknown and stochastic environments.

4.3 Problem Definition: Subtask Graph Inference Problem

We formulate the *subtask graph inference* problem, an instance of few-shot RL problem where a task is parameterized by *subtask graph* [Sohn et al., 2018]. The details of how a subtask graph parameterizes the MDP is described in Appendix A.1. Our problem extends the subtask graph *execution* problem in [Sohn et al., 2018] by removing the assumption that a subtask graph is given to the agent; thus, the agent must infer the subtask graph in order to perform the complex task. Following few-shot RL settings, the agent's goal is to quickly adapt to the given task (*i.e.*, MDP) in the adaptation phase to maximize the return in the test phase (see Figure 4.1). A task consists of N subtasks and the subtask graph models a hierarchical dependency between subtasks.

Subtask: A subtask Φ^i can be defined by a tuple (*completion set* $\mathcal{S}^i_{\text{comp}} \subset \mathcal{S}$, *precondition* $G^i_{\mathbf{c}} : \mathcal{S} \mapsto \{0, 1\}$, *subtask reward function* $G^i_{\mathbf{r}} : \mathcal{S} \to \mathbb{R}$). A subtask Φ^i is *complete* if the current state is contained in its completion set (*i.e.*, $\mathbf{s}_t \in \mathcal{S}^i_{\text{comp}}$), and the agent receives a reward $r_t \sim G^i_{\mathbf{r}}$ upon the completion of subtask Φ^i. A subtask Φ^i is *eligible* (*i.e.*, subtask can be executed)

Require: $p(G)$: distribution over subtask graph

while not done **do**
  Sample batch of task parameters $\{G_i\}_{i=1}^{M} \sim p(G)$
  for all G_i in the batch **do**
    Rollout K episodes $\tau = \{\mathbf{s}_t, \mathbf{o}_t, r_t, d_t\}_{t=1}^{H} \sim \pi_\theta^{\text{adapt}}$ in task $\mathcal{M}_{G_i}$ ▷ adaptation phase
    Compute r_t^{UCB} as in Eq.(4.6)
    $\hat{G}_i = \text{ILP}(\tau)$ ▷ subtask graph inference
    Sample $\tau' \sim \pi_{\hat{G}_i}^{\text{exe}}$ in task $\mathcal{M}_{G_i}$ ▷ test phase
  end for
  Update $\theta \leftarrow \theta + \eta \nabla_\theta \sum_{i=1}^{M} \mathcal{R}_{\mathcal{M}_{G_i}}^{\text{PG+UCB}} \left(\pi_\theta^{\text{adapt}}\right)$ using $\mathcal{R}_{\mathcal{M}}^{\text{PG+UCB}}$ in Eq.(4.8)
end while

Program 4.1: Adaptation policy optimization during meta-training

if its precondition $G_{\mathbf{c}}^i$ is satisfied (see Figure 4.1 for examples). A subtask graph is a tuple of precondition and subtask reward of all the subtasks: $G = (G_{\mathbf{c}}, G_{\mathbf{r}})$. Then, the task defined by the subtask graph is a factored MDP [Boutilier et al., 1995, Schuurmans and Patrascu, 2002]; *i.e.*, the transition model is factored as $p(\mathbf{s}'|\mathbf{s}, a) = \prod_i p_{G_{\mathbf{c}}^i}(s_i'|\mathbf{s}, a)$ and the reward function is factored as $R(\mathbf{s}, a) = \sum_i R_{G_{\mathbf{r}}{}^i}(\mathbf{s}, a)$ (see Appendix for the detail). The main benefit of factored MDP is that it allows us to model many hierarchical tasks in a principled way with a compact representation such as dynamic Bayesian network [Dean and Kanazawa, 1989, Boutilier et al., 1995]. For each subtask Φ^i, the agent can learn an option $\mathcal{O}^i$ [Sutton et al., 1999b] that *executes* the subtask[2].

Environment: The state input to the agent at time step t consists of $\mathbf{s}_t = \{\mathbf{x}_t, \mathbf{e}_t, \text{step}_t, \text{epi}_t, \mathbf{obs}_t\}$.

- **Completion**: $\mathbf{x}_t \in \{0, 1\}^N$ indicates whether each subtask is complete.
- **Eligibility**: $\mathbf{e}_t \in \{0, 1\}^N$ indicates whether each subtask is eligible (*i.e.*, precondition is satisfied).
- **Time budget**: $\text{step}_t \in \mathbb{R}$ is the remaining time steps until episode termination.
- **Episode budget**: $\text{epi}_t \in \mathbb{R}$ is the remaining number of episodes in adaptation phase.
- **Observation**: $\mathbf{obs}_t \in \mathbb{R}^{\mathcal{H} \times \mathcal{W} \times \mathcal{C}}$ is a (visual) observation at time t.

At time step t, we denote the option taken by the agent as $\mathbf{o}_t$ and the binary variable that indicates whether episode is terminated as d_t.

Require: The current parameter θ
Require: A task $\mathcal{M}_G$ parametrized by a task parameter G (unknown to the agent)
1: Roll out K train episodes $\tau_H = \{\mathbf{s}_t, \mathbf{o}_t, r_t, d_t\}_{t=1}^{H} \sim \pi_\theta^{\text{adapt}}$ in task $\mathcal{M}_G$ ▷ adaptation phase
2: Infer a subtask graph: $\hat{G} = (\hat{G}_{\mathbf{c}}, \hat{G}_{\mathbf{r}}) = (\text{ILP}(\tau_H), \text{RI}(\tau_H))$ ▷ task inference
3: Roll out a test episode $\tau' = \{\mathbf{s}'_t, \mathbf{o}'_t, r'_t, d'_t\}_{t=1}^{H'} \sim \pi_{\hat{G}}^{\text{exe}}$ in task $\mathcal{M}_G$ ▷ test phase
4: Measure the performance $R = \sum_t r'_t$ for this task

Program 4.2: Process of single trial for a task $\mathcal{M}_G$ at meta-test time

4.4 Method

We propose a *Meta-learner with Subtask Graph Inference* (MSGI) which infers the latent subtask graph G. Figure 4.1 overviews our approach. Our main idea is to employ two policies: adaptation policy and test policy. During the adaptation phase, an *adaptation policy* $\pi_\theta^{\text{adapt}}$ rolls out K episodes of *adaptation trajectories*. From the collected adaptation trajectories, the agent infers the subtask graph $\hat{G}$ using inductive logic programming (ILP) technique. A *test policy* $\pi_{\hat{G}}^{\text{test}}$, conditioned on the inferred subtask graph $\hat{G}$, rolls out episodes and maximizes the return in the test phase. Note that the performance depends on the quality of the inferred subtask graph. The adaptation policy indirectly contributes to the performance by improving the quality of inference. Intuitively, if the adaptation policy completes more diverse subtasks during adaptation, the more "training data" is given to the ILP module, which results in more accurate inferred subtask graph. Algorithm 6.1 summarizes our meta-training procedure. For meta-testing, see Algorithm 4.2.

4.4.1 Subtask Graph Inference

Let $\tau_H = \{\mathbf{s}_1, \mathbf{o}_1, r_1, d_1, \ldots, \mathbf{s}_H\}$ be an adaptation trajectory of the adaptation policy $\pi_\theta^{\text{adapt}}$ for K episodes (or H steps in total) in adaptation phase. The goal is to infer the subtask graph G for this task, specified by preconditions $G_{\mathbf{c}}$ and subtask rewards $G_{\mathbf{r}}$. We find the maximum-likelihood estimate (MLE) of $G = (G_{\mathbf{c}}, G_{\mathbf{r}})$ that maximizes the likelihood of the adaptation trajectory τ_H: $\hat{G}^{\text{MLE}} = \arg\max_{G_{\mathbf{c}}, G_{\mathbf{r}}} p(\tau_H | G_{\mathbf{c}}, G_{\mathbf{r}})$.

[2]As in Andreas et al. [2017], Oh et al. [2017], Sohn et al. [2018], such options are pre-learned with curriculum learning; the policy is learned by maximizing the subtask reward, and the initiation set and termination condition are given as $\mathcal{I}^i = \{\mathbf{s} | G_{\mathbf{c}}^i(\mathbf{s}) = 1\}$ and $\beta^i = \mathbb{I}(x^i = 1)$

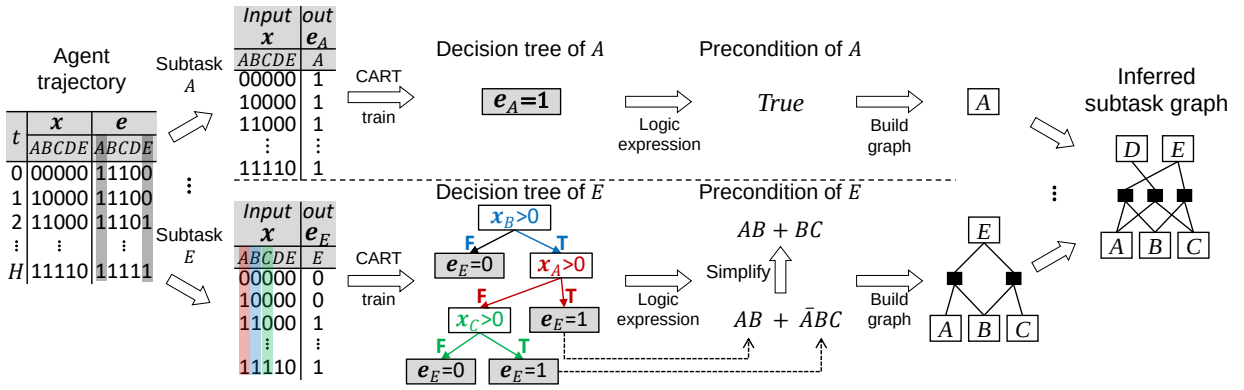


Figure 4.2: Our inductive logic programming module infers the precondition $G_{\mathbf{c}}$ from adaptation trajectory. For example, the decision tree of subtask *E* (bottom row) estimates the latent precondition function $f_{G_{\mathbf{c}}^E} : \mathbf{x} \mapsto \mathbf{e}^E$ by fitting its input-output data (i.e., agent's trajectory $\{\mathbf{x}_t, \mathbf{e}_t^E\}_{t=1}^H$). The decision tree is constructed by choosing a variable (i.e., a component of $\mathbf{x}$) at each node that best splits the data. The learned decision trees of all the subtasks are represented as logic expressions, and then transformed and merged to form a subtask graph.

The likelihood term can be expanded as

$$p(\tau_H|G_{\mathbf{c}}, G_{\mathbf{r}}) = p(\mathbf{s}_1|G_{\mathbf{c}}) \prod_{t=1}^{H} \pi_\theta\left(\mathbf{o}_t|\tau_t\right) p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{o}_t, G_{\mathbf{c}}) p(r_t|\mathbf{s}_t, \mathbf{o}_t, G_{\mathbf{r}}) p(d_t|\mathbf{s}_t, \mathbf{o}_t) \tag{4.1}$$

$$\propto p(\mathbf{s}_1|G_{\mathbf{c}}) \prod_{t=1}^{H} p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{o}_t, G_{\mathbf{c}}) p(r_t|\mathbf{s}_t, \mathbf{o}_t, G_{\mathbf{r}}), \tag{4.2}$$

where we dropped the terms that are independent of G. From the definitions in Section 4.3, precondition $G_{\mathbf{c}}$ defines the mapping $\mathbf{x} \mapsto \mathbf{e}$, and the subtask reward $G_{\mathbf{r}}$ determines the reward as $r_t \sim {G_{\mathbf{r}}}^i$ if subtask i is eligible (*i.e.*, $\mathbf{e}_t^i = 1$) and option $\mathcal{O}^i$ is executed at time t. Therefore, we have

$$\hat{G}^{\text{MLE}} = (\hat{G}_{\mathbf{c}}^{\text{MLE}}, \hat{G}_{\mathbf{r}}^{\text{MLE}}) = \left(\arg\max_{G_{\mathbf{c}}} \prod_{t=1}^{H} p(\mathbf{e}_t|\mathbf{x}_t, G_{\mathbf{c}}),\ \arg\max_{G_{\mathbf{r}}} \prod_{t=1}^{H} p(r_t|\mathbf{e}_t, \mathbf{o}_t, G_{\mathbf{r}}) \right). \tag{4.3}$$

We note that no supervision from the ground-truth subtask graph G is used. Below we explain how to compute the estimate of preconditions $\hat{G}_{\mathbf{c}}^{\text{MLE}}$ and subtask rewards $\hat{G}_{\mathbf{r}}^{\text{MLE}}$.

Precondition inference via logic induction Since the precondition function $f_{G_{\mathbf{c}}} : \mathbf{x} \mapsto \mathbf{e}$ (see Section 4.3 for definition) is a deterministic mapping, the probability term $p(\mathbf{e}_t|\mathbf{x}_t, G_{\mathbf{c}})$ in Eq.(4.3)

is 1 if $\mathbf{e}_t = f_{G_\mathbf{c}}(\mathbf{x}_t)$ and 0 otherwise. Therefore, we can rewrite $\hat{G}^{\text{MLE}}_{\mathbf{c}}$ in Eq.(4.3) as:

$$\hat{G}^{\text{MLE}}_{\mathbf{c}} = \arg\max_{G_\mathbf{c}} \prod_{t=1}^{H} \mathbb{I}(\mathbf{e}_t = f_{G_\mathbf{c}}(\mathbf{x}_t)), \tag{4.4}$$

where $\mathbb{I}(\cdot)$ is the indicator function. Since the eligibility $\mathbf{e}$ is factored, the precondition function $f_{G_{\mathbf{c}^i}}$ for each subtask is inferred independently. We formulate the problem of finding a boolean function that satisfies all the indicator functions in Eq.(4.4) (*i.e.*, $\prod_{t=1}^{H} \mathbb{I}(\mathbf{e}_t = f_{G_\mathbf{c}}(\mathbf{x}_t)) = 1$) as an *inductive logic programming* (ILP) problem [Muggleton, 1991]. Specifically, $\{\mathbf{x}_t\}_{t=1}^{H}$ forms binary vector inputs to programs, and $\{e^i_t\}_{t=1}^{H}$ forms Boolean-valued outputs of the i-th program that denotes the eligibility of the i-th subtask. We use the *classification and regression tree* (CART) to infer the precondition function $f_{G_\mathbf{c}}$ for each subtask based on Gini impurity [Breiman, 1984]. Intuitively, the constructed decision tree is the simplest boolean function approximation for the given input-output pairs $\{\mathbf{x}_t, \mathbf{e}_t\}$. Then, we convert it to a logic expression (*i.e.*, precondition) in sum-of-product (SOP) form to build the subtask graph. Figure 4.2 summarizes the overall logic induction process.

Subtask reward inference To infer the subtask reward function $\hat{G}^{\text{MLE}}_{\mathbf{r}}$ in Eq.(4.3), we model each component of subtask reward as a Gaussian distribution $G^i_\mathbf{r} \sim \mathcal{N}(\hat{\mu}^i, \hat{\sigma}^i)$. Then, $\hat{\mu}^i_{\text{MLE}}$ becomes the empirical mean of the rewards received after taking the eligible option $\mathcal{O}^i$ in the trajectory τ_H:

$$\hat{G}^{\text{MLE},i}_{\mathbf{r}} = \hat{\mu}^i_{\text{MLE}} = \mathbb{E}\left[r_t | \mathbf{o}_t = \mathcal{O}^i, \mathbf{e}^i_t = 1\right] = \frac{\sum_{t=1}^{H} r_t \mathbb{I}(\mathbf{o}_t = \mathcal{O}^i, \mathbf{e}^i_t = 1)}{\sum_{t=1}^{H} \mathbb{I}(\mathbf{o}_t = \mathcal{O}^i, \mathbf{e}^i_t = 1)}. \tag{4.5}$$

4.4.2 Test phase: Subtask Graph Execution Policy

Once a subtask graph $\hat{G}$ has been inferred, we can derive a *subtask graph execution (SGE) policy* $\pi^{\text{exe}}_{\hat{G}}(\mathbf{o}|\mathbf{x})$ that aims to maximize the cumulative reward in the test phase. Note that this is precisely the problem setting used in Sohn et al. [2018]. Therefore, we employ a graph reward propagation (GRProp) policy [Sohn et al., 2018] as our SGE policy. Intuitively, the GRProp policy approximates a subtask graph to a differentiable form such that we can compute the gradient of modified return with respect to the completion vector to measure how much each subtask is likely to increase the modified return.

4.4.3 Learning: Optimization of the Adaptation Policy

We now describe how to learn the adaptation policy $\pi^{\text{adapt}}_\theta$, or its parameters θ. We can directly optimize the objective $\mathcal{R}_{\mathcal{M}_G}(\pi)$ using policy gradient methods [Williams, 1992, Sutton et al., 1999a], such as actor-critic method with generalized advantage estimation (GAE) [Schulman et al.,

2016]. However, we find it challenging to train our model for two reasons: 1) delayed and sparse reward (*i.e.*, the return in the test phase is treated as if it were given as a one-time reward at the last step of adaptation phase), and 2) large task variance due to highly expressive power of subtask graph. To facilitate learning, we propose to give an intrinsic reward r_t^{UCB} to agent in addition to the extrinsic environment reward, where r_t^{UCB} is the upper confidence bound (UCB) [Auer et al., 2002]-inspired exploration bonus term as follows:

$$r_t^{\text{UCB}} = w_{\text{UCB}} \cdot \mathbb{I}(\mathbf{x}_t \text{ is novel}), \quad w_{\text{UCB}} = \sum_{i=1}^{N} \frac{\log(n^i(0) + n^i(1))}{n^i(e_t^i)}, \tag{4.6}$$

where N is the number of subtasks, e_t^i is the eligibility of subtask i at time t, and $n^i(e)$ is the visitation count of e^i (i.e., the eligibility of subtask i) during the adaptation phase until time t. The weight w_{UCB} is designed to encourage the agent to make eligible and execute those subtasks that have infrequently been eligible, since such rare data points in general largely improve the inference by balancing the dataset that CART (i.e., our logic induction module) learns from. The conditioning term $\mathbb{I}(\mathbf{x}_t \text{ is novel})$ encourages the adaptation policy to visit novel states with a previously unseen completion vector $\mathbf{x}_t$ (*i.e.*, different combination of completed subtasks), since the data points with same $\mathbf{x}_t$ input will be ignored in the ILP module as a duplication. We implement $\mathbb{I}(\mathbf{x}_t \text{ is novel})$ using a hash table for computational efficiency. Then, the intrinsic objective is given as follows:

$$\mathcal{R}_{\mathcal{M}_G}^{\text{UCB}}\left(\pi_\theta^{\text{adapt}}\right) = \mathbb{E}_{\pi_\theta^{\text{adapt}}, \mathcal{M}_G}\left[\sum_{t=1}^{H} r_t^{\text{UCB}}\right], \tag{4.7}$$

where H is the horizon of adaptation phase. Finally, we train the adaptation policy $\pi_\theta^{\text{adapt}}$ using an actor-critic method with GAE [Schulman et al., 2016] to maximize the following objective:

$$\mathcal{R}_{\mathcal{M}_G}^{\text{PG+UCB}}\left(\pi_\theta^{\text{adapt}}\right) = \mathcal{R}_{\mathcal{M}_G}\left(\pi_{\hat{G}}^{\text{GRProp}}\right) + \beta_{\text{UCB}} \mathcal{R}_{\mathcal{M}_G}^{\text{UCB}}\left(\pi_\theta^{\text{adapt}}\right), \tag{4.8}$$

where $\mathcal{R}_{\mathcal{M}_G}(\cdot)$ is the meta-learning objective in Eq.(2.4), β_{UCB} is the mixing hyper-parameter, and $\hat{G}$ is the inferred subtask graph that depends on the adaptation policy $\pi_\theta^{\text{adapt}}$. The complete procedure for training our MSGI agent with UCB reward is summarized in Algorithm 6.1.

4.5 Experiments

In the experiment, we investigate the following research questions:

1. Does MSGI correctly infer task parameters G?
2. Does adaptation policy $\pi_\theta^{\text{adapt}}$ improve the efficiency of few-shot RL?

3. Does the use of UCB bonus facilitate training? (§ 4.5.3)

4. How well does MSGI perform compared with other meta-RL algorithms?

5. Can MSGI generalize to longer adaptation horizon, and unseen and more complex tasks?

We evaluate our approach in comparison with the following baselines:

- Random is a policy that executes a random eligible subtask that has not been completed.
- RL^2 is the meta-RL agent in Duan et al. [2016], trained to maximize the return over K episodes.
- HRL is the hierarchical RL agent in Sohn et al. [2018] trained with the same actor-critic method as our approach during adaptation phase. The network parameter is reset when the task changes.
- GRProp+Oracle is the GRProp policy [Sohn et al., 2018] provided with the ground-truth subtask graph as input. This is roughly an upper bound of the performance of MSGI-based approaches.
- MSGI-Rand (Ours) uses a random policy as an adaptation policy, with the task inference module.
- MSGI-Meta (Ours) uses a meta-learned policy (*i.e.*, $\pi_\theta^{\text{adapt}}$) as an adaptation policy, with the task inference module.

For RL^2 and HRL, we use the same network architecture as our MSGI adaptation policy. The domains on which we evaluate these approaches include two simple grid-world environments (**Mining** and **Playground**) [Sohn et al., 2018] and a more challenging domain **SC2LE** [Vinyals et al., 2017] (StarCraft II).

4.5.1 Experiments on Mining and Playground Domains

Mining [Sohn et al., 2018] is inspired by Minecraft (see Figure 4.3) where the agent receives reward by picking up raw materials in the world or crafting items with raw materials. **Playground** [Sohn et al., 2018] is a more flexible and challenging domain, where the environment is stochastic and subtask graphs are *randomly* generated (*i.e.*, precondition is an arbitrary logic expression). We follow the setting in Sohn et al. [2018] for choosing train/evaluation sets. We measure the performance in terms of normalized reward $\hat{R} = (R - R_{\text{min}})/(R_{\text{max}} - R_{\text{min}})$ averaged over 4 random seeds, where R_{min} and R_{max} correspond to the average reward of the Random and the GRProp+Oracle agent, respectively.

4.5.1.1 Training Performance

Figure 4.4 shows the learning curves of MSGI-Meta and RL^2, trained on the **D1**-Train set of **Playground** domain. We set the adaptation budget in each trial to $K = 10$ episodes. For MSGI-Rand and HRL (which are not meta-learners), we show the average performance after 10 episodes

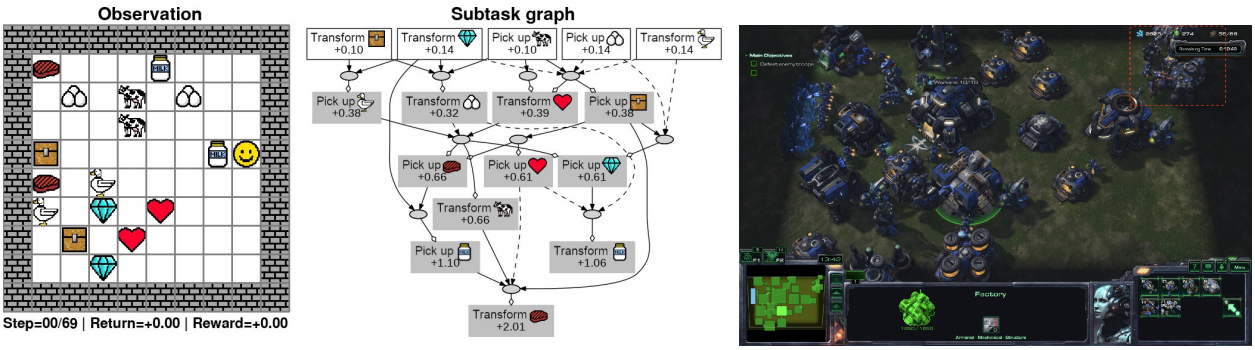


Figure 4.3: **Left**: A visual illustration of **Playground** domain and an example of underlying subtask graph. The goal is to execute subtasks in the optimal order to maximize the reward within time budget. The subtask graph describes subtasks with the corresponding rewards (e.g., transforming a chest gives 0.1 reward) and dependencies between subtasks through AND and OR nodes. For instance, the agent must first *transform chest* AND *transform diamond* before executing *pick up duck*. **Right**: A *warfare* scenario in **SC2LE** domain [Vinyals et al., 2017]. The agent must prepare for the upcoming warfare by training appropriate units, through an appropriate order of subtasks (see Appendix for more details).

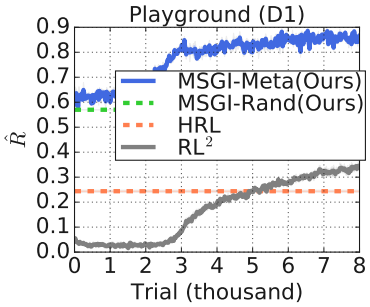


Figure 4.4: Learning curves on the **Playground** domain. We measure the normalized reward (y-axis) in a test phase, after a certain number of training trials (x-axis).

of adaptation. As training goes on, the performance of MSGI-Meta significantly improves over MSGI-Rand with a large margin. It demonstrates that our meta adaptation policy learns to explore the environment more efficiently, inferring subtask graphs more accurately. We also observe that the performance of RL^2 agent improves over time, eventually outperforming the HRL agent. This indicates that RL^2 learns 1) a good initial policy parameter that captures the common knowledge generally applied to all the tasks and 2) an efficient adaptation scheme such that it can adapt to the given task more quickly than standard policy gradient update in HRL.

4.5.1.2 Adaptation and Generalization Performance

Adaptation efficiency. In Figure 4.5, we measure the test performance (in terms of the normalized reward $\hat{R}$) by varying episode budget K (*i.e.*, how many episodes are used in adaptation phase), after

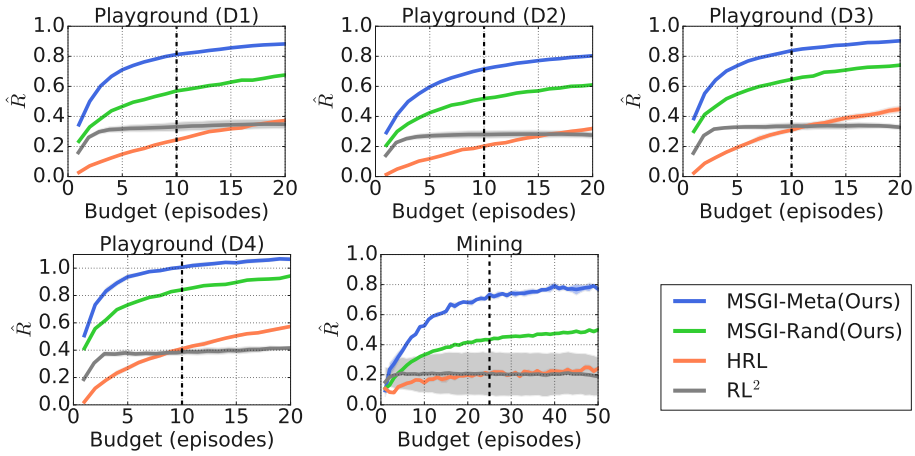


Figure 4.5: Generalization performance on unseen tasks (**D1**-Eval, **D2, D3, D4**, and **Mining**-Eval) with varying adaptation horizon. We trained agent with the fixed adaptation budget ($K = 10$ for **Playground** and $K = 25$ for **Mining**) denoted by the vertical dashed line, and tested with varying unseen adaptation budgets. We report the average normalized return during test phase, where GRProp+Oracle is the upper bound (*i.e.*, $\hat{R} = 1$) and Random is the lower bound (*i.e.*, $\hat{R} = 0$). The shaded area in the plot indicates the range between $\hat{R} + \sigma$ and $\hat{R} - \sigma$ where σ is the standard error of normalized return.

8000 trials of meta-training (Figure 4.4). Intuitively, it shows how quickly the agent can adapt to the given task. Our full algorithm MSGI-Meta consistently outperforms MSGI-Rand across all the tasks, showing that our meta adaptation policy can efficiently explore informative states that are likely to result in more accurate subtask graph inference. Also, both of our MSGI-based models perform better than HRL and RL^2 baselines in all the tasks, showing that explicitly inferring underlying task structure and executing the predicted subtask graph is more effective than learning slow-parameters and fast-parameters (*e.g.*, RNN states) on those tasks involving complex subtask dependencies.

Generalization performance. We test whether the agents can generalize over unseen task and longer adaptation horizon, as shown in Figure 4.5. For **Playground**, we follow the setup of [Sohn et al., 2018]: we train the agent on **D1**-Train with the adaptation budget of 10 episodes, and test on unseen graph distributions **D1**-Eval and larger graphs **D2-D4** (See Appendix A.2 for more details about the tasks in Playground and Mining). We report the agent's performance as the normalized reward with up to 20 episodes of adaptation budget. For **Mining**, the agent is trained on randomly generated graphs with 25 episodes budget and tested on 440 hand-designed graphs used in [Sohn et al., 2018], with up to 50 episodes of adaptation budget. Both of our MSGI-based models generalize well to unseen tasks and over different adaptation horizon lengths, continuingly improving the agent's performance. It demonstrates that the efficient exploration scheme that our meta adaptation policy can generalize to unseen tasks and longer adaptation horizon, and that our task execution policy, GRProp, generalizes well to unseen tasks as already shown in [Sohn et al.,

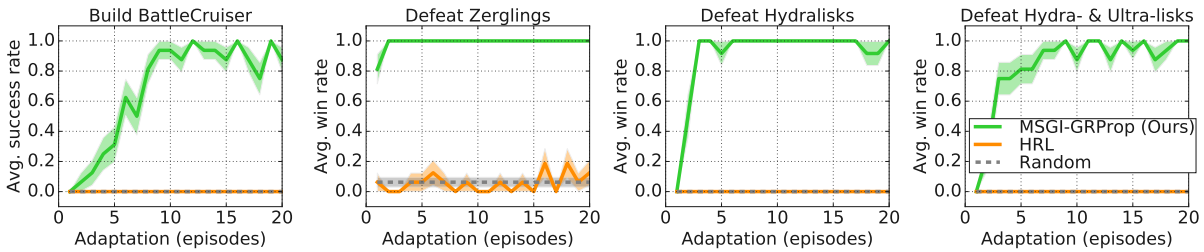


Figure 4.6: Adaptation performance with different adaptation horizon on **SC2LE** domain.

2018]. However, RL^2 fails to generalize to unseen task and longer adaptation horizon: on **D2-D4** with adaptation horizons longer than the length the meta-learner was trained for, the performance of the RL^2 agent is almost stationary or even decreases for very long-horizon case (**D2, D3**, and **Mining**), eventually being surpassed by the HRL agent. This indicates (1) the adaptation scheme that RL^2 learned does not generalize well to longer adaptation horizons, and (2) a common knowledge learned from the training tasks does not generalize well to unseen test tasks.

4.5.2 Experiments on StarCraft II Domain

SC2LE [Vinyals et al., 2017] is a challenging RL domain built upon the real-time strategy game StarCraft II. We focus on two particular types of scenarios: *Defeat Enemy* and *Build Unit*. Each type of the scenarios models the different aspect of challenges in the full game. The goal of *Defeat Enemy* is to eliminate various enemy armies invading within 2,400 steps. We consider three different combinations of units with varying difficulty: *Defeat Zerglings, Defeat Hydralisks, Defeat Hydralisks & Ultralisks* (see Figure B.1 in Appendix B.1 and demo videos at `https://bit.ly/msgi-videos`). The goal of *Build Unit* scenario is to build a specific unit within 2,400 steps. To showcase the advantage of MSGI infering the underlying subtask graph, we set the target unit as *Battlecruiser*, which is at the highest rank in the technology tree of *Terran* race. In both scenarios, the agent needs to train the workers, collect resources, and construct buildings and produce units in correct sequential order to win the game. Each building or unit has a precondition as per the technology tree of the player's race (see Appendix B.1 for more details).

Agents. Note that the precondition of each subtask is determined by the domain and remains fixed across the tasks. If we train the meta agents (MSGI-Meta and RL^2), the agents memorize the subtask dependencies (*i.e.*, over-fitting) and does not learn any useful policy for efficient adaptation. Thus, we only evaluate Random and HRL as our baseline agents. Instead of MSGI-Meta, we used MSGI-GRProp. MSGI-GRProp uses the GRProp policy as an adaptation policy since GRProp is a good approximation algorithm that works well without meta-training as shown in [Sohn et al., 2018]. Since the environment does not provide any subtask-specific reward, we set the subtask

reward using the UCB bonus term in Eq. (4.6) to encourage efficient exploration.

Subtask graph inference. We quantitatively evaluate the inferred subtask graph in terms of the precision and recall of the inferred precondition function $f_{\hat{\mathbf{c}}} : \mathbf{x} \mapsto \hat{\mathbf{e}}$. Specifically, we compare the inference output $\hat{\mathbf{e}}$ with the GT label $\mathbf{e}$ generated by the GT precondition function $f_{\mathbf{c}} : \mathbf{x} \mapsto \mathbf{e}$ for all possible binary assignments of input (i.e., completion vector $\mathbf{x}$). For all the tasks, our MSGI-GRProp agent almost perfectly infers the preconditions with more than 94% precision and 96% recall of all possible binary assignments, when averaged over all 163 preconditions in the game, with only 20 episodes of adaptation budget. We provide the detailed quantitative and qualitative results on the inferred subtask graph in supplemental material.

Adaptation efficiency. Figure 4.6 shows the adaptation efficiency of MSGI-GRProp, HRL agents, and Random policy on the four scenarios. We report the average victory or success rate over 8 episodes. MSGI-GRProp consistently outperforms HRL agents with a high victory rate, by (1) quickly figuring out the useful units and their prerequisite buildings and (2) focusing on executing these subtasks in a correct order. For example, our MSGI-GRProp learns from the inferred subtask graph that some buildings such as *sensor tower* or *engineering bay* are unnecessary for training units and avoids constructing them.

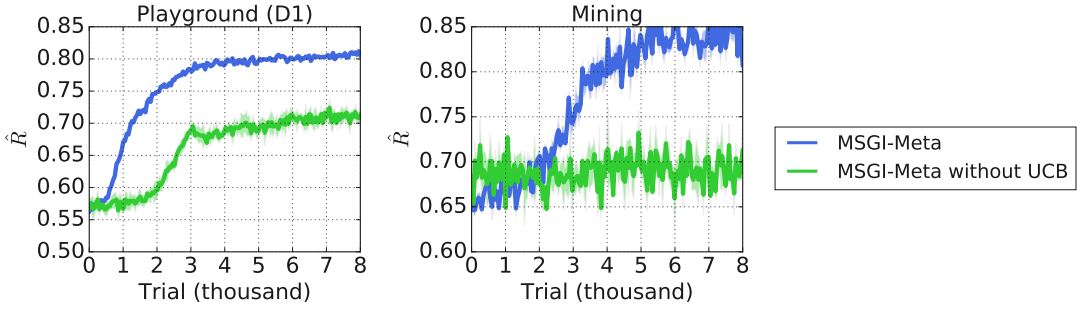


Figure 4.7: Comparison of meta-training MSGI-Meta agent that was trained with UCB bonus and extrinsic reward, and MSGI-Meta without UCB agent that was trained with extrinsic reward only in the **Playground** and **Mining** domain. In both domains, adding UCB bonus improves the meta-training performance of our MSGI-Meta agent.

4.5.3 Ablation study on the intrinsic reward

We conducted an ablation study comparing our MSGI-Meta with and without UCB bonus. We will refer our method with UCB bonus as MSGI-Meta, and our method without UCB bonus as MSGI-Meta `without UCB`. Figure 4.7 shows that UCB bonus facilitates the meta-training of our MSGI-Meta agents in both **Playground** and **Mining** domains.

4.6 Discussion

We introduced and addressed a few-shot RL problem with a complex subtask dependencies. We proposed to learn the adaptation policy that efficiently collects experiences in the environment, infer the underlying hierarchical task structure, and maximize the expected reward using the execution policy given the inferred subtask graph. The empirical results confirm that our agent can efficiently explore the environment during the adaptation phase that leads to better task inference and leverage the inferred task structure during the test phase. In this work, we assumed that the option is pre-learned and the environment provides the status of each subtask. In the future work, our approach may be extended to more challenging settings where the relevant subtask structure is fully learned from pure observations, and options to execute these subtasks are also automatically discovered.

CHAPTER 5

Fast Inference and Transfer of Compositional Task Structures for Few-shot Task Generalization

This chapter proposes a novel method that can learn a prior model of task structure from the training tasks and transfer it to the unseen tasks for fast adaptation. We formulate this as a few-shot reinforcement learning problem where a task is characterized by a subtask graph which describes a set of subtasks and their dependencies that are unknown to the agent. Instead of directly inferring an unstructured task embedding, our multi-task subtask graph inferencer (MTSGI) infers the common task structure in terms of the subtask graph from the training tasks, and use it as a prior to improve the task inference in testing. To this end, we propose to model the prior sampling and posterior update for the subtask graph inference. Our experiment results on 2D grid-world and complex web navigation domains show that the proposed method can learn and leverage the common underlying structure of the tasks for faster adaptation to the unseen tasks than various existing algorithms such as meta reinforcement learning, hierarchical reinforcement learning, and other heuristic agents.

5.1 Introduction

Recently, deep reinforcement learning (RL) has shown an outstanding performance on various domains such as video games [Mnih et al., 2015, Vinyals et al., 2019] and board games [Silver et al., 2017]. However, most of the successes of deep RL were focused on a single-task setting where the agent is allowed to interact with the environment for hundreds of millions of time steps. In numerous real-world scenarios, interacting with the environment is expensive or limited, and the agent is often presented with a novel task that is not seen during its training time. In order to overcome this limitation, many recent works focused on scaling the RL algorithm beyond the single task setting. Recent works on multi-task RL aim to build a single, contextual policy that can solve multiple related tasks and generalize to unseen tasks. However, they require a certain form of task embedding as an extra input that often fully characterizes the given task [Oh et al., 2017,

Andreas et al., 2017, Yu et al., 2017, Chaplot et al., 2018], or requires a human demonstration Huang et al. [2018], which are not readily available in practice. Meta RL [Finn et al., 2017, Duan et al., 2016] focuses on a more general setting where the agent should learn about the unseen task purely via interacting with environment without any additional information. However, such meta-RL algorithms either require a large amount of experience on the sufficiently diverse set of tasks or are limited to a relatively smaller set of simple tasks with similar task structures.

On the contrary, real-world problems require the agent to solve much more complex and compositional tasks without human supervision. Consider a web-navigating RL agent given the task of checking out the products from an online store. The agent can complete the task by filling out the correct information to the corresponding web elements such as shipping or payment information, navigating between the web pages, and finally, placing the order by clicking the appropriate button. Note that the task consists of multiple *subtasks* and the subtasks have complex dependencies in the form of *precondition*; For instance, the agent may proceed to the payment web page *after* all the required shipping information has been correctly filled in, or the `credit_card_number` field will show up *after* selecting the `credit_card` as a payment method. Learning to perform such task can be quite challenging when the reward is given only after yielding meaningful outcomes (*i.e.*, sparse reward task). This is the problem scope we focus on in this work: solving compositional tasks with complex subtask dependencies and generalizing to unseen tasks without human supervision.

Recent works [Sohn et al., 2019, Xu et al., 2017, Huang et al., 2018, Liu et al., 2016, Ghazanfari and Taylor, 2017] tackled the compositional tasks by explicitly inferring the underlying task structure in a graph form. Specifically, the subtask graph inference (SGI) framework [Sohn et al., 2019] uses inductive logic programming (ILP) on the agent's own experience to infer the task structure in terms of *subtask graph* and learns a contextual policy to *execute* the inferred task in few-shot RL setting. However, it only meta-learned the adaptation policy that relates to the efficient exploration, while the task inference and execution policy learning were limited to a single task (*i.e.*, both task inference and policy learning were done from scratch for each task), limiting its capability of handling large variance in the task structure. We claim that the inefficient task inference may hinder applying the SGI framework to a more complex domain such as web navigation [Shi et al., 2017, Liu et al., 2018] where a task may have a large number of subtasks and complex dependencies between them. We note that human can navigate an unseen website by transferring the high-level process learned from previously seen websites.

Inspired by this, we extend the SGI framework to a *multi-task subtask graph inferencer* (MTSGI) that can generalize the previously learned high-level task structure to the unseen task for faster adaptation and stronger generalization. MTSGI estimates the prior model of the subtask graphs inferred from the training tasks. When an unseen task is presented, MTSGI samples the prior that best matches with the current task, and incorporate the sampled prior model to improve the

inference of posterior over the latent subtask graph, which in turn improves the policy learning. We demonstrate results in the 2D grid-world domain and the web navigation domain that simulates the interaction with ten actual websites. We compare our method with MSGI [Sohn et al., 2019] that learns the task hierarchy from scratch for each task, and two other baselines including hierarchical RL and a heuristic algorithm. We find that our agent significantly outperforms all other baselines, and the result demonstrates that the prior model learned by MTSGI enables more efficient task inference compared to MSGI.

5.2 Related Work

Meta-reinforcement learning. Our work aims to tackle the few-shot reinforcement learning (RL) problem via multi-task task inference. Researchers have previously studied meta-reinforcement learning in order to solve the few-shot reinforcement learning problem, where there exist two broad categories of meta-RL approaches: RNN and gradient-based methods. The RNN-based meta-RL methods [Duan et al., 2016, Wang et al., 2016, Hochreiter et al., 2001] encode the common knowledges of the task into the hidden states and the parameters of the RNN. The gradient-based meta-RL methods [Finn et al., 2017, Nichol et al., 2018, Gupta et al., 2018, Finn et al., 2018, Kim et al., 2018] encode the task embedding in terms of the initial policy parameter for fast adaptation through meta gradient. Existing meta-RL approaches, however, often require a large amount of environment interaction due to the long-horizon nature of the few-shot RL tasks. Our work instead explicitly infers the underlying task parameter in terms of subtask graph, which can be efficiently inferred using inductive logic programming (ILP) method and be transferred across different, unseen tasks.

Multi-task reinforcement learning. Multi-task reinforcement learning aims to learn an inductive bias that can be shared and used across a variety of related RL tasks to improve the task generalization. Early works mostly focused on the transfer learning oriented approaches [Lazaric, 2012, Taylor and Stone, 2009] such as instance transfer [Lazaric et al., 2008] or representation transfer [Konidaris and Barto, 2006]. However, these algorithms rely heavily on the prior knowledge about the allowed task differences. Hausman et al. [2018], Pinto and Gupta [2017], Wilson et al. [2007] proposed to train a multi-task policy with multiple objectives from different tasks. However, the gradients from different tasks may conflict and hurt the training of other tasks. To avoid gradient conflict, Zhang and Yeung [2014], Chen et al. [2018], Lin et al. [2019] proposed to explicitly model the task similarity. However, dynamically modulating the loss or the gradient of RL update often results in the instability in optimization. Our multi-task learning algorithm also takes the transfer learning oriented viewpoint; MTSGI captures and transfers the task knowledge in terms of the subtask graph. However, our work does not make a strong assumption on the task distribution.

We only assume that the task is parameterized by unknown subtask graph, which subsumes many existing compositional tasks (*e.g.*, Oh et al. [2017], Andreas et al. [2017], Huang et al. [2018], etc).

Web navigating RL agent. Previous work introduced MiniWoB [Shi et al., 2017] and MiniWoB++ [Liu et al., 2018] benchmarks that are manually curated sets of simulated toy environments for the web navigation problem. They formulated the problem as acting on a page represented as a Document Object Model (DOM), a hierarchy of objects in the page. The agent is trained with human demonstrations and online episodes in an RL loop. Jia et al. [2019] proposed a graph neural network based DOM encoder and a multi-task formulation of the problem similar to this work. Gur et al. [2018] introduced a manually-designed curriculum learning method and an LSTM based DOM encoder. DOM level representations of web pages pose a significant sim-to-real gap as simulated websites are considerably smaller (100s of nodes) compared to noisy real websites (1000s of nodes). As a result, these models are trained and evaluated on the same simulated environments which is difficult to deploy on real websites. Our work formulates the problem as abstract web navigation on real websites where the objective is to learn a latent subtask dependency graph similar to sitemap of websites. We propose a multi-task training objective that generalizes from a fixed set of real websites to unseen websites without any demonstration, illustrating an agent capable of navigating real websites for the first time.

5.3 Subtask Graph Inference Problem

The *subtask graph inference* problem [Sohn et al., 2019] is a few-shot RL problem where a task is parameterized by a set of subtasks and their dependencies. Formally, a task consists of N subtasks $\{\Phi^1, \dots, \Phi^N\}$, and each subtask Φ^i is parameterized by a tuple $({\mathcal{S}_{\text{comp}}}^i, G^i_{\mathbf{c}}, G^i_{\mathbf{r}})$. The *goal state* ${\mathcal{S}_{\text{comp}}}^i$ and *precondition* $G^i_{\mathbf{c}}$ defines the condition that a subtask is *completed*: the current state should be contained in its goal states (*i.e.*, $\mathbf{s}_t \in \mathcal{S}^i_{\text{comp}}$) and the precondition should be satisfied (*i.e.*, $f_c(\mathbf{s}_t) = 1$). If the precondition is not satisfied, the subtask cannot be completed and agent receives no reward even if the goal state is achieved. The *subtask reward function* $G^i_{\mathbf{r}}$ defines the amount of reward given to the agent when it *completes* the subtask i: $r_t \sim G^i_{\mathbf{r}}$. We note that the subtasks $\{\Phi^1, \dots, \Phi^N\}$ are unknown to the agent. Thus, the agent should learns to infer the underlying task structure and complete the subtasks in an optimal order while satisfying the required preconditions.

State. In subtask graph inference problem, it is assumed that the state input provides high-level status of the subtasks at each time step. Intuitively, this enables the agent to infer the latent subtask structure only from the interaction with environment. Specifically, the state consists of the followings: $\mathbf{s}_t = (\text{obs}_t, \mathbf{x}_t, \mathbf{e}_t, \text{step}_{\text{epi},t}, \text{step}_{\text{phase},t})$. The $\text{obs}_t \in \{0,1\}^{W \times H \times C}$ is a visual observation of the environment. The completion vector $\mathbf{x}_t \in \{0,1\}^N$ indicates whether each subtask is complete. The eligibility vector $\mathbf{e}_t \in \{0,1\}^N$ indicates whether each subtask is eligible (*i.e.*, precondition is

Require: π^{adapt}: adaptation policy, $\mathcal{T}^{\text{prior}}$: prior set
1: $\mathcal{T}^{\text{prior}} \leftarrow \emptyset$
2: **for** each task $\mathcal{M}_i \in \mathcal{M}^{\text{train}}$ and its subtask set Φ^i **do**
3: Rollout adaptation phase for K steps:
4: $\tau = \{\mathbf{s}_t, \mathbf{o}_t, r_t, d_t\}_{t=1}^{K} \sim \pi^{\text{adapt}}$ in task $\mathcal{M}_i$.
5: Infer subtask graph $G^{\text{MLE}} = \arg\max_G p(\tau|G)$
6: $\pi^{\text{exe}} = \text{GRProp}(G^{\text{MLE}})$
7: Rollout test phase: $\tau^{\text{test}} \sim \pi^{\text{exe}}$ in task $\mathcal{M}_i$
8: Update prior $\mathcal{T}^{\text{prior}} \leftarrow \mathcal{T}^{\text{prior}} \cup \left(G^{\text{MLE}}, \Phi^i, R(\tau^{\text{test}})\right)$
9: **end for**

Program 5.1: Meta-training: learning the prior

satisfied). Following the few-shot RL setting, the agent observes two scalar-valued time features: the remaining time steps until the episode termination $\text{step}_{\text{epi},t} \in \mathbb{R}$ and the remaining time steps until the phase termination $\text{step}_{\text{phase},t} \in \mathbb{R}$.

Options. For each subtask Φ^i, the agent can learn an option $\mathcal{O}^i$ [Sutton et al., 1999b] that reaches the goal state of the subtask. Following Sohn et al. [2019], such options are pre-learned individually by maximizing the goal-reaching reward: $r_t = \mathbb{I}(\mathbf{s}_t \in \mathcal{S}^i_{\text{comp}})$. At time step t, we denote the option taken by the agent as $\mathbf{o}_t$ and the binary variable that indicates whether episode is terminated as d_t.

5.4 Method

We propose a novel multi-task learning extension of subtask graph inference framework (MTSGI) that can perform an efficient posterior inference of latent task embedding (*i.e.*, subtask graph). Specifically, MTSGI infers the subtask graph of the training tasks $\mathcal{M}^{\text{train}}$ and estimates the prior model from the inferred subtask graphs during meta-training. During meta-evaluation, MTSGI first samples the subtask graph from the prior model, and incorporates the agent's adaptation trajectory to update the posterior estimate of the latent subtask graph of the evaluation tasks $\mathcal{M}^{\text{eval}}$.

5.4.1 Meta-training: learning the prior

Let τ be an adaptation trajectory of the agent for K steps. The goal is to infer the latent subtask graph G for the given training task $\mathcal{M}_G \in \mathcal{M}^{\text{train}}$, specified by preconditions $G_{\mathbf{c}}$ and subtask rewards $G_{\mathbf{r}}$. We find the maximum-likelihood estimate (MLE) of $G = (G_{\mathbf{c}}, G_{\mathbf{r}})$ that maximizes the likelihood of the adaptation trajectory τ:

$$\hat{G}^{\text{MLE}} = \arg\max_{G_{\mathbf{c}}, G_{\mathbf{r}}} p(\tau|G_{\mathbf{c}}, G_{\mathbf{r}}). \tag{5.1}$$

Following Sohn et al. [2019], we infer the precondition $G_{\mathbf{c}}$ and the subtask reward $G_{\mathbf{r}}$ as follows:

$$\hat{G}_{\mathbf{c}}^{\text{MLE}} = \arg\max_{G_{\mathbf{c}}} \prod_{t=1}^{K} \mathbb{I}(\mathbf{e}_t = f_{G_{\mathbf{c}}}(\mathbf{x}_t)), \tag{5.2}$$

$$\hat{G}_{\mathbf{r}}^{\text{MLE},i} = (\hat{\mu}_{\text{MLE}}^{i}, \hat{\sigma}_{\text{MLE}}^{i}), \tag{5.3}$$

$$\hat{\mu}_{\text{MLE}}^{i} = \mathbb{E}\left[r_t | \mathbf{o}_t = \mathcal{O}^i, \mathbf{e}_t^i = 1\right], \tag{5.4}$$

$$\hat{\sigma^2}_{\text{MLE}}^{i} = \mathbb{E}\left[(r_t - \hat{\mu}_{\text{MLE}}^{i})^2 | \mathbf{o}_t = \mathcal{O}^i, \mathbf{e}_t^i = 1\right], \tag{5.5}$$

where $\mathbf{e}_t$ is the eligibility vector, $\mathbf{x}_t$ is the completion vector, $\mathbf{o}_t$ is the option taken by the agent, r_t is the reward at time step t, and $\mathcal{O}^i$ is the option corresponding to the i-th subtask. See Section 4.4.1 for the detailed derivation.

Precondition inference. The problem in Equation (5.2) is known as the inductive logic programming (ILP) problem that finds a boolean function that satisfies all the indicator functions. Specifically, $\{\mathbf{x}_t\}_{t=1}^{H}$ forms binary vector inputs to programs, and $\{e_t^i\}_{t=1}^{H}$ forms Boolean-valued outputs of the i-th program that denotes the eligibility of the i-th subtask. We use the *classification and regression tree* (CART) to infer the precondition function $f_{G_{\mathbf{c}}}$ for each subtask based on Gini impurity [Breiman, 1984]. Intuitively, the constructed decision tree is the simplest boolean function approximation for the given input-output pairs $\{\mathbf{x}_t, \mathbf{e}_t\}$. The decision tree is converted to a logic expression (*i.e.*, precondition) in sum-of-product (SOP) form to build the subtask graph.

Subtask reward inference. In Equation (5.3), the MLE of the subtask reward $\hat{G}_{\mathbf{r}}^{\text{MLE},i}$ is given as the empirical mean and variance of the option reward of each subtask. Thus, we accumulate the reward over the option execution and store them during the adaptation. In test phase, we compute the empirical mean and variance from the recorded rewards and use them as the Gaussian parameters of subtask reward distribution.

For all the training tasks, we store the inferred subtask graph $\hat{G}^{\text{MLE}}$ (*i.e.*, the precondition and the subtask reward) after the adaptation. The stored subtask graphs are used as a prior model (See Section 5.4.3) in meta-evaluation. The overall process of meta-training is summarized in **Algorithm** 5.1.

5.4.2 Adaptation policy

The goal of adaptation policy is to maximally explore and gather the information about the task to accurately infer the latent subtask graph. Intuitively, adaptation policy should try to make diverse subtasks eligible and complete since the precondition $G_{\mathbf{c}}$ is inferred from the completion and

Require: π^{exe}: test policy (GRProp), π^{adapt}: adaptation policy, $\mathcal{T}^{\text{prior}}$: prior set
1: **for** each task $\mathcal{M}_i \in \mathcal{M}^{\text{eval}}$ and its subtask set Φ^i **do**
2: Sample prior: $(G^{\text{prior}}, \Phi^{\text{prior}}, R(\tau^{\text{prior}})) \sim p(\mathcal{T}^{\text{prior}})$
3: $\hat{G}^{\text{prior}} = \text{Map}_{\Phi^{\text{prior}} \to \Phi^i}(G^{\text{prior}})$ ▷ subtask mapping
4: Rollout adaptation phase for K steps:
5: $\tau = \{\mathbf{s}_t, \mathbf{o}_t, r_t, d_t\}_{t=1}^{K} \sim \pi^{\text{adapt}}$ in task $\mathcal{M}_i$
6: Infer subtask graph $G^{\text{MLE}} = \arg\max_G p(\tau|G)$
7: $\pi^{\text{exe}} \propto \text{GRProp}(G^{\text{MLE}})^{\alpha} \cdot \text{GRProp}(\hat{G}^{\text{prior}})^{(1-\alpha)}$
8: Rollout test phase: $\tau^{\text{test}} \sim \pi^{\text{exe}}$ in task $\mathcal{M}_i$
9: **end for**

Program 5.2: Meta-evaluation: multi-task SGI

eligibility pair and the subtask reward $G_{\mathbf{r}}$ can be inferred from the option reward of the completed subtasks. Inspired by Sohn et al. [2019][1], we used the soft-version of upper confidence bound (UCB) [Auer et al., 2002] exploration policy as adaptation policy as follows:

$$\pi_{\text{adapt}}\left(o = \mathcal{O}^i \mid s\right) \propto \exp\left(r_i + \frac{\log\left(\sum_j n_j\right)}{\sqrt{2} n_i}\right), \tag{5.6}$$

where r_i is the the empirical mean of the reward received after executing subtask i and n_i is the number of times subtask i has been executed within current task. During meta-training, we reset the parameters $\{r_i, n_i\}_{i=1}^{N}$ to zero in the beginning of each task, and after the adaptation is over (*i.e.*, $t = K$) the final parameters $\{r_i, n_i\}_{i=1}^{N}$ were stored together with the inferred subtask graph as prior data. During meta-evaluation, we initialized the current parameters $\{r_i, n_i\}_{i=1}^{N}$ with those of the sampled prior task to incorporate the information gathered in the sampled prior task. Intuitively, the agent is encouraged to execute the subtask that was either not present in the prior task (*i.e.*, $n_i = 0$) or have been executed less often (*i.e.*, $n_i \ll \sum_j n_j$) than others such that the agent has relatively less information about the subtask.

5.4.3 Meta-evaluation: prior sampling

After constructing the prior model, our MTSGI chooses the subtask from the prior model that is most similar to the given evaluation task. Specifically, we define the pair-wise similarity between a prior task $\mathcal{M}_G^{\text{prior}}$ and the current evaluation task $\mathcal{M}_G$ as follows:

$$\text{sim}\left(\mathcal{M}_G, \mathcal{M}_G^{\text{prior}}\right) = F_\beta\left(\Phi, \Phi^{\text{prior}}\right) + \kappa R\left(\tau^{\text{prior}}\right), \tag{5.7}$$

[1] Sohn et al. [2019] used the hard-version of UCB policy as a teacher policy for pretraining the adaptation policy.

where F_β is the F-score with weight parameter β, Φ is the subtask set of $\mathcal{M}_G$, Φ^{prior} is the subtask set of $\mathcal{M}_G^{\text{prior}}$, and $R\left(\tau^{\text{prior}}\right)$ is the agent's empirical performance on the prior task $\mathcal{M}_G^{\text{prior}}$. F_β measures how many subtasks overlap between current and prior tasks in terms of precision and recall as follows:

$$F_\beta = \left(1+\beta^2\right) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}, \tag{5.8}$$

$$\text{Precision} = |\Phi \cap \Phi^{\text{prior}}|/|\Phi^{\text{prior}}|, \tag{5.9}$$

$$\text{Recall} = |\Phi \cap \Phi^{\text{prior}}|/|\Phi|. \tag{5.10}$$

We used $\beta = 10$ to assign higher weight to current task (*i.e.*, recall) than the prior task (*i.e.*, precision). The pseudo-code of the prior sampling process is summarized in **Algorithm** 5.2.

5.4.4 Meta-evaluation: multi-task subtask graph inference

Let τ be the adaptation trajectory of the current task $\mathcal{M}_G$, and $\mathcal{T}^{\text{p}} = \{\tau_1^{\text{p}}, \ldots, \tau_N^{\text{p}}\}$ be the adaptation trajectories of the N (seen) training tasks. Assuming that $p(G)$ follows the uniform distribution, our multi-task policy is parametrized by the subtask graph G as follows:

$$\pi(o_t|s_t, \tau, \mathcal{T}^{\text{p}}) = \sum_G \pi(o_t|s_t, G) p(G|\tau, \mathcal{T}^{\text{p}}) \tag{5.11}$$

$$\propto \sum_G \pi(o_t|s_t, G) p(\tau|G, \mathcal{T}^{\text{p}}) p(G|\mathcal{T}^{\text{p}}) \tag{5.12}$$

$$\propto \sum_G \pi(o_t|s_t, G) p(\tau|G, \mathcal{T}^{\text{p}}) p(\mathcal{T}^{\text{p}}|G) p(G) \tag{5.13}$$

$$\propto \sum_G \pi(o_t|s_t, G) p(\tau|G, \mathcal{T}^{\text{p}}) p(\mathcal{T}^{\text{p}}|G) \tag{5.14}$$

$$= \sum_G \pi(o_t|s_t, G)^\alpha p(\tau|G, \mathcal{T}^{\text{p}}) \pi(o_t|s_t, G)^{(1-\alpha)} p(\mathcal{T}^{\text{p}}|G) \tag{5.15}$$

$$\simeq \left\{\sum_G \pi(o_t|s_t, G) p(\tau|G, \mathcal{T}^{\text{p}})\right\}^\alpha \cdot \left\{\sum_G \pi(o_t|s_t, G) p(\mathcal{T}^{\text{p}}|G)\right\}^{(1-\alpha)} \tag{5.16}$$

$$\simeq \pi(o_t|s_t, G_\tau)^\alpha \pi(o_t|s_t, G^{\text{prior}})^{(1-\alpha)} \tag{5.17}$$

where $G_\tau = \arg\max_G p(\tau|G, \mathcal{T}^{\text{p}})$ is the the maximum likelihood estimate (MLE) of the current subtask graph G given the entire experience and $G^{\text{prior}} = \arg\max_G p(\mathcal{T}^{\text{p}}|G)$ is the MLE of the subtask graph G given the prior experience $\mathcal{T}^{\text{p}}$. For efficient computation, we first compute and store

the MLE of all the training tasks G^{prior} during meta-training. In meta-evaluation, we sample G^{prior} together with the prior, and merge with the current policy as in Equation (5.17). The pseudo-code of multi-task subtask graph inference process is summarized in **Algorithm** 5.2.

Subtask Graph Setting			
Task	#Subtasks	#Distractors	Episode length
Amazon	31	4	27
Apple	43	5	40
BestBuy	37	6	37
Converse	42	6	43
Dick's	39	6	37
eBay	39	5	37
Ikea	39	5	37
Samsung	42	6	41
Target	39	6	47
Walmart	46	5	43

Table 5.1: The task configuration of the tasks in **SymWoB** domain. Each task is parameterized by different subtask graphs, and the episode length is manually set according to the challengeness of the tasks.

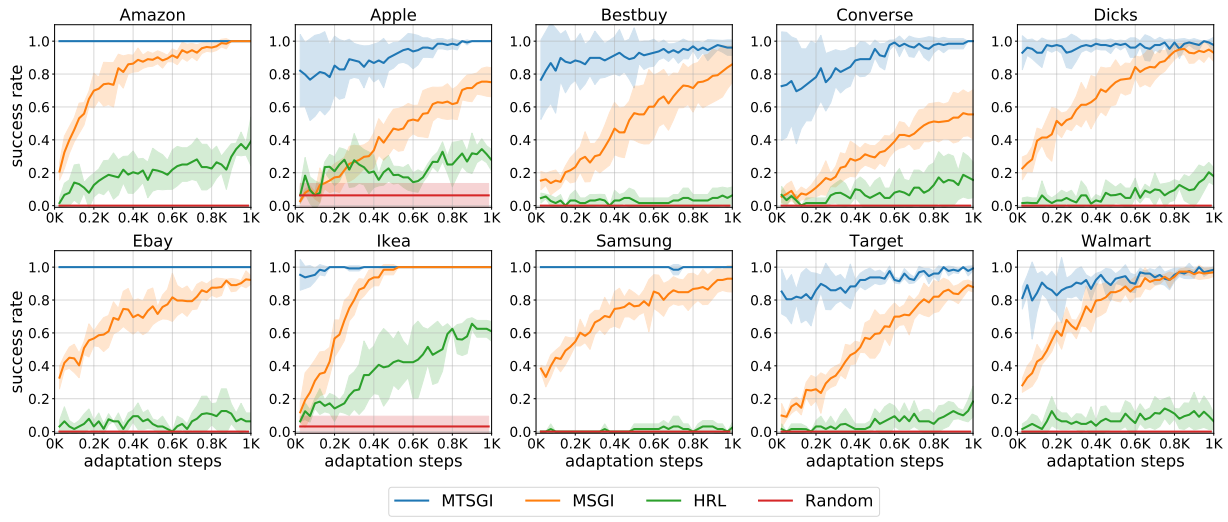


Figure 5.1: The success rate of the compared methods in the test phase in terms of the environment step during adaptation phase on **SymWoB** domain.

5.5 Experiment

5.5.1 Domain

We evaluate our method on 2D grid-world domain and realistic symbolic web navigation domain.

5.5.1.1 Mining

Mining [Sohn et al., 2018] is inspired by Minecraft game where the agent receives reward by picking up raw materials in the world or crafting items with raw materials. The subtask dependency in **Mining** domain comes from the crafting recipe implemented in the game. Following Sohn et al. [2018], we used the four set of pre-generated training/testing task splits generated with four different random seeds. Each split set consists of 3200 training tasks and 440 testing tasks for meta-training and meta-evaluation, respectively. The performance was averaged over the four task split sets.

5.5.1.2 SymWoB

We implement a symbolic version of the checkout process on the 10 real-world websites: **Amazon**, **Apple**, **BestBuy**, **Converse**, **Dick's**, **eBay**, **Ikea**, **Samsung**, **Target**, and **Walmart**. The details of the tasks are summarized in Table 5.1.

Subtasks. Each actionable web element (*e.g.*, text field, button, radio button, drop-down list, and hyperlink) is considered as a subtask, and the agent can execute the subtask by selecting an option corresponding to the subtask.

Reward function and episode termination. The agent may receive a non-zero reward only at the end of episode (*i.e.*, sparse-reward task). When the episode terminates due to the time budget, the agent may not receive any reward. Otherwise, the following two types of subtasks terminate the episode and give a non-zero reward upon completion:

- **Goal subtask** is considered as a success and the agent receives +5 reward, and the episode is terminated. In our checkout tasks, the `Click_Place_Order` subtask is the goal subtask.
- **Distractor subtask** does not contribute to solving the given task but terminates the episode with -1 reward. It models the web elements that lead to external web pages such as clicking `Help` or `Leave_Feedback` buttons.

Transition dynamics. The transition dynamics follows the dynamics of the actual website. Each website consists of multiple web pages. The agent may only execute the subtasks that are currently visible (*i.e.*, on the current web page) and can navigate to the next web page only after filling out all the required field and clicking the continue button. The `Click_Place_Order` button is present

in the last web page. Therefore, the agent must learn to navigate through the web pages in order to finish the current task.

Completion and eligibility. For each subtask, the completion and eligibility is computed based on the status of the corresponding web element. For example, the subtask corresponding to a text field is *completed* if the text field is filled with the correct profile information, and it is *eligible* if the text field is visible on the website and has not yet been completed.

For more details about each task, please refer to Appendix C.1.

5.5.2 Agents

We compared the following algorithms in the experiment.

- MTSGI (Ours): our multi-task SGI agent
- MSGI [Sohn et al., 2019]: SGI agent without multi-task learning
- HRL: an Option [Sutton et al., 1999b]-based proximal policy optimization (PPO) [Schulman et al., 2017] agent with the gated rectifier unit (GRU)
- Random: a heuristic policy that uniform randomly executes an eligible subtask

Training. We used the ten websites for meta-evaluation. For each website, we randomly sampled a single website among the remaining nine websites and used it for meta-training. For example, we meta-trained our MTSGI on **Amazon** and tested on **Samsung**. The RL agents (*e.g.*, HRL) were individually trained on each testing website; the policy was initialized when a new task is sampled and trained during the adaptation phase. All the experiments were repeated with four random seeds, where different training tasks were sampled for different random seed.

5.5.3 Result: few-shot generalization performance

Figure 5.1 and Figure 5.2 show the few-shot generalization performance of the compared methods on **SymWoB** and **Mining**. MTSGI achieves close to 100% zero-shot success rate (*i.e.*, success rate at x-axis=0) on five out of ten websites, which is significantly higher than the zero-shot performance of MSGI. This indicates that the prior learned from randomly chosen website significantly improves the subtask graph inference and in turn improves the multi-task test policy. Moreover, our MTSGI can learn a near-optimal policy on the remaining five websites after only 1,000 steps of environment interactions, demonstrating that the proposed multi-task learning scheme enables the fast adaptation. Even though the MSGI agent is learning each task from scratch, it still outperforms the HRL and Random agents, showing that explicitly inferring the underlying task structure and executing the predicted subtask graph is significantly more effective than learning the policy from the reward signal (*i.e.*, HRL) on those tasks involving complex subtask dependencies. Given the pre-learned

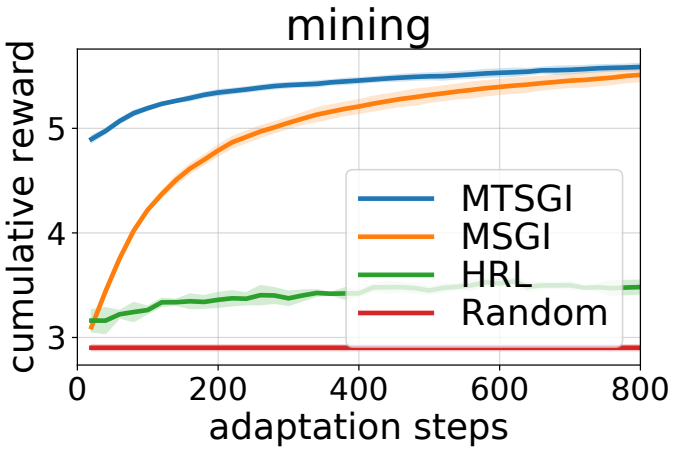


Figure 5.2: The cumulative reward of the compared methods in the test phase in terms of the environment step during adaptation phase on **Mining** domain.

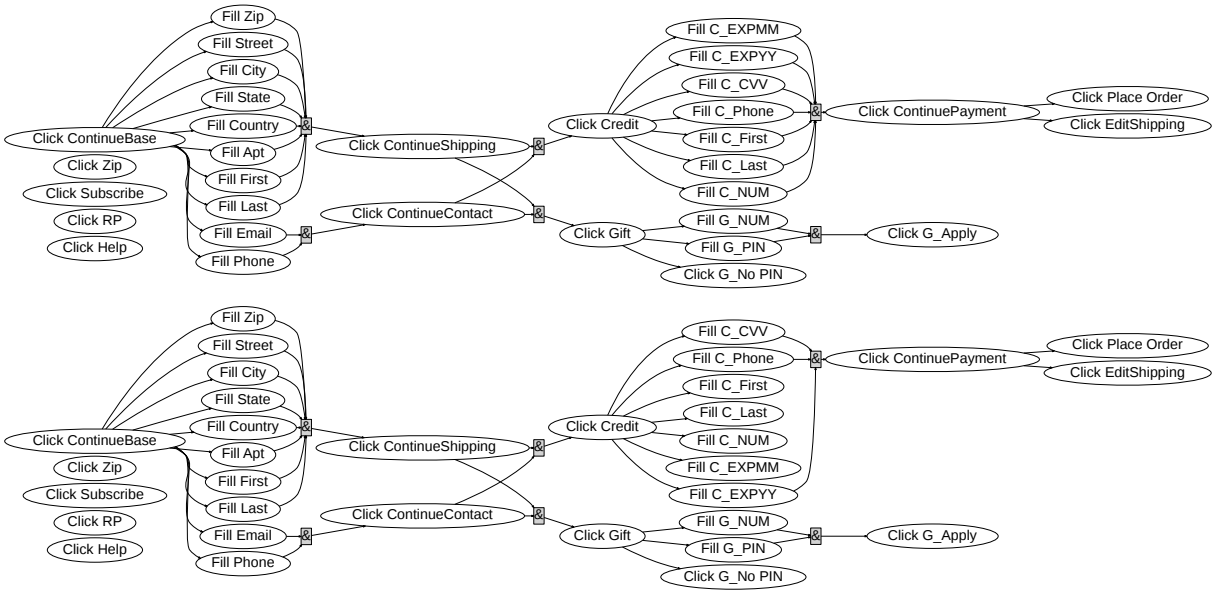


Figure 5.3: (Top) The ground-truth subtask graph of **Walmart** domain (not available to the agent) and (Bottom) The subtask graph inferred by our MTSGI after 1,000 steps of environment interaction on **Walmart** domain.

options, HRL agent improves the success rate during the adaptation by updating the high-level policy. However, training the policy requires a large amount of interactions especially for the tasks with many distractors (*e.g.*, **eBay**, **Samsung**, and **BestBuy**, etc).

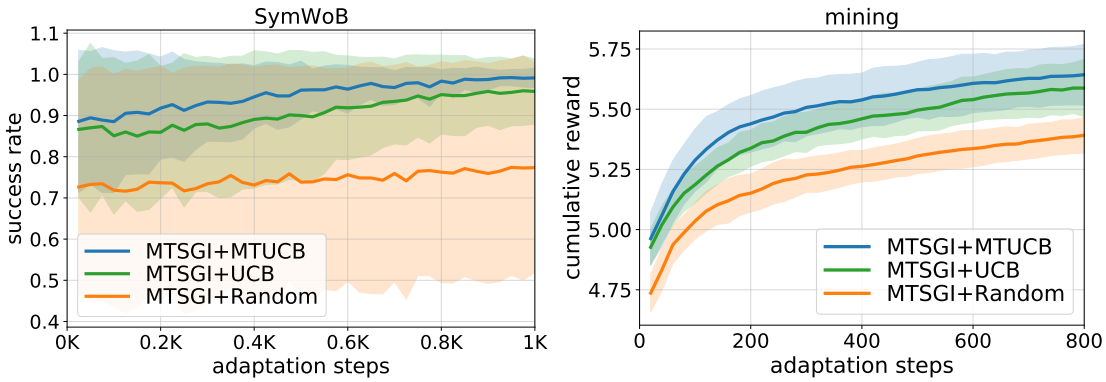


Figure 5.4: Comparison of different exploration strategies for MTSGI used in adaptation phase for **SymWoB** and **Mining** domains. The performance was averaged over the ten websites for **SymWoB**. We report the mean (solid curve) and standard error (shadowed area) of the performance over four random seeds.

5.5.4 Analysis on the inferred subtask graph

We evaluate the accuracy of the subtask graph inference process of the MTSGI by comparing the inferred subtask graph against the ground-truth subtask graph. Note that the ground-truth subtask graph is not available to the agent in both meta-train and meta-evaluation stage; Instead, it is inferred purely from the agent's trajectory.

Figure 5.3 shows the ground-truth subtask graph and the subtask graph inferred by MTSGI in **Walmart** domain after 1,000 steps of adaptation. We can see that MTSGI can infer the subtask graph quite accurately; the inferred subtask graph is missing only four preconditions of `Click_Continue_Payment` subtasks, and for all other subtasks, their preconditions are correctly inferred. We note that such small error in the subtask graph has negligible effect as shown in Figure 5.1:*i.e.*, MTSGI achieves near-optimal performance on **Walmart** after 1,000 steps of adaptation.

5.5.5 Ablation study: effect of exploration strategy in adaptation phase

In this section, we investigate the effect of various exploration strategies on the performance of MTSGI. We compared the following three adaptation policies:

- MTSGI+Random: Random policy that uniformly randomly execute any eligible subtask is used as an adaptation policy for both meta-training and meta-testing.
- MTSGI+UCB: The UCB policy (See §5.4.2) that aims to execute the novel subtask is used as an adaptation policy for both meta-training and meta-testing. The counts of subtasks execution is initialized when a new task is sampled.

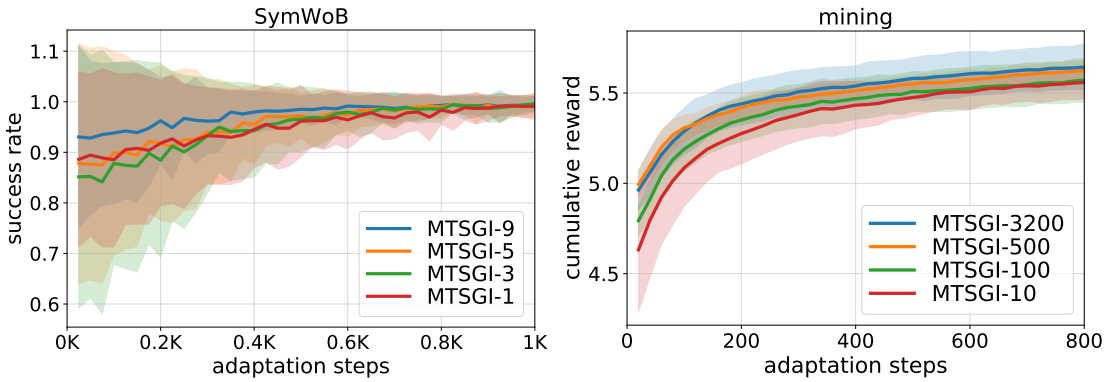


Figure 5.5: Comparison of different number of priors and its effect on the performance on both **SymWoB** and **Mining** domains.

- MTSGI+MTUCB: We use UCB policy for meta-training and store the subtask execution counts as a part of prior. When the prior is sampled in meta-testing, we *transfer* the prior knowledge by initializing the subtask execution counts with those of the sampled prior.

Figure 5.4 summarizes the result on **SymWoB** and **Mining** domain, respectively. Using the more sophisticated exploration policy such as MTSGI+UCB or MTSGI+MTUCB improved the performance of MTSGI compared to MTSGI+Random, which was also observed in Sohn et al. [2019]. This is because the better exploration helps the adaptation policy collect more data for logic induction by executing more diverse subtasks. In turn, this results in more accurate subtask graph inference and better performance. Also, MTSGI+MTUCB outperforms MTSGI+UCB on both domains. This indicates that transferring the exploration counts makes the agent's exploration more efficient in evaluation tasks. Intuitively, the transferred exploration counts inform the agent which subtasks were *under-explored* during meta-training, such that the agent can focus more on gathering the information that are missing in the prior during meta-evaluation.

5.5.6 Ablation study: effect of the training task set size

MTSGI learns the prior from the training tasks. We investigated how many training tasks are required for MTSGI to learn a good prior for transfer learning. We compared the performance of MTSGI with the varying number of training tasks: 1, 3, 5, 9 tasks for **SymWoB** and 10, 100, 500, 3200 tasks for **Mining** in Figure 5.5. The training tasks are randomly subsampled from the largest training set. The result shows that training on larger number of tasks consistently improved the performance, but saturates at some point (*e.g.*, $|\mathcal{M}^{\text{train}}| = 9$ for **SymWoB** and $|\mathcal{M}^{\text{train}}| = 500$ for **Mining**). **Mining** generally requires more number of training tasks than **SymWoB** because the agent is required to perform 440 tasks in **Mining** while **SymWoB** was evaluated on only 10 tasks;

agent is required to capture a wider range of task distribution in **Mining** than **SymWoB**. Also, we note that MTSGI can still adapt much more efficiently than all other baseline methods even when only a small number of training tasks are available (*e.g.*, one task for **SymWoB** and ten tasks for **Mining**).

5.6 Discussion

We introduce multi-task RL extension of the subtask graph inference framework that can quickly adapt to the unseen tasks by modeling the prior of subtask graph from the training tasks and transferring it to the test tasks. Specifically, our multi-task subtask graph inferencer (MTSGI) samples a prior by measuring the similarity between the current task and the prior task and merges the two policies constructed from the subtask graph inferred from prior and current tasks, respectively. The empirical results demonstrate that our MTSGI achieves strong zero- and few-shot generalization performance on 2D grid-world and complex web navigation domains by transferring the common knowledges learned in the training tasks to the unseen ones in terms of subtask graph.

In this work, we have assumed that the subtasks and the corresponding options are pre-learned and that the environment provides a high-level status of each subtask (*e.g.*, whether the web element is filled in with the correct information). In the future work, our approach may be extended to a more general setting where the relevant subtask structure is fully learned from pure observations, and the corresponding options are also automatically discovered as well.

CHAPTER 6

Shortest-Path Constrained Reinforcement Learning for Sparse Reward Tasks

This work proposes the k-Shortest-Path (k-SP) constraint: a novel constraint on the agent's trajectory that improves the sample-efficiency in sparse-reward MDPs. We show that any optimal policy necessarily satisfies the k-SP constraint. Notably, the k-SP constraint prevents the policy from exploring state-action pairs along the non-k-SP trajectories (*e.g.*, going back and forth). However, in practice, excluding state-action pairs may hinder convergence of RL algorithms. To overcome this, we propose a novel cost function that penalizes the policy violating SP constraint, instead of completely excluding it. Our numerical experiment in a tabular RL setting demonstrate that the SP constraint can significantly reduce the trajectory space of policy. As a result, our constraint enables more sample efficient learning by suppressing redundant exploration and exploitation. Our experiments on *MiniGrid*, *DeepMind Lab* and *Atari* show that the proposed method significantly improves proximal policy optimization (PPO) and outperforms existing novelty-seeking exploration methods including count-based exploration, indicating that it improves the sample efficiency by preventing the agent from taking redundant actions.

6.1 Introduction

Recently, deep reinforcement learning (RL) has achieved a large number of breakthroughs in many domains including video games [Mnih et al., 2015, Vinyals et al., 2019], and board games [Silver et al., 2017]. Nonetheless, a central challenge in reinforcement learning (RL) is the sample efficiency [Kakade et al., 2003]; it has been shown that the RL algorithm requires a large number of samples for successful learning in MDP with large state and action space. Moreover, the success of RL algorithm heavily hinges on the quality of collected samples; the RL algorithm tends to fail if the collected trajectory does not contain enough evaluative feedback (*e.g.*, sparse or delayed reward).

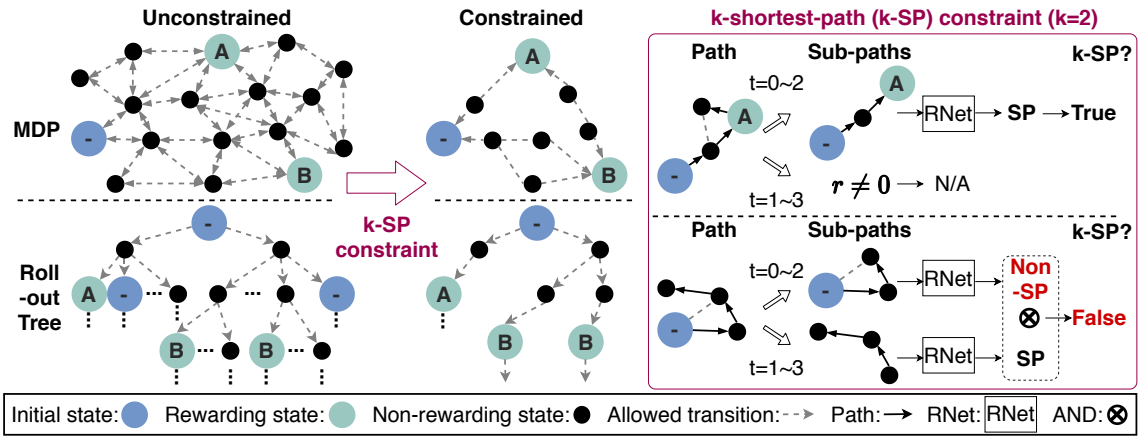


Figure 6.1: The k-SP constraint improves sample efficiency of RL methods in sparse-reward tasks by pruning out suboptimal trajectories from the trajectory space. Intuitively, the k-SP constraint means that when a policy rolls out into trajectories, all of sub-paths of length k is a shortest path (under a distance metric defined in terms of policy, discount factor and transition probability; see Section 6.3.2 for the formal definition). **(Left)** MDP and a rollout tree are given. **(Middle)** The paths that satisfy the k-SP constraint. The number of admissible trajectories is drastically reduced. **(Right)** A path rolled out by a policy satisfies the k-SP constraint if all sub-paths of length k are shortest paths and have not received non-zero reward. We use a reachability network to test if a given (sub-)path is a shortest path (See Section 6.4 for details).

To circumvent this challenge, planning-based methods utilize the environment's model to improve or create a policy instead of interacting with environment. Recently, combining the planning method with an efficient path search algorithm, such as Monte-Carlo tree search (MCTS) [Norvig, 2002, Coulom, 2006], has demonstrated successful results [Guo et al., 2016, Vodopivec et al., 2017, Silver et al., 2017]. However, such tree search methods would require an accurate model of MDP and the complexity of planning may grow intractably large for complex domain. Model-based RL methods attempt to learn a model instead of assuming that model is given, but learning an accurate model also requires a large number of samples, which is often even harder to achieve than solving the given task. Model-free RL methods can be learned solely from the environment reward, without the need of a (learned) model. However, both value-based and policy-based methods suffer from poor sample efficiency especially in sparse-reward tasks. To tackle sparse reward problems, researchers have proposed to learn an intrinsic bonus function that measures the novelty of the state that agent visits [Schmidhuber, 1991, Oudeyer and Kaplan, 2009, Pathak et al., 2017, Savinov et al., 2018b, Choi et al., 2018, Burda et al., 2018]. However, when such intrinsic bonus is added to the reward, it often requires a careful balancing between environment reward and bonus and scheduling of the bonus scale in order to guarantee the convergence to optimal solution.

To tackle aforementioned challenge of sample efficiency in sparse reward tasks, we introduce a constrained-RL framework that improves the sample efficiency of any model-free RL algorithm

in sparse-reward tasks, under the mild assumptions on MDP (see Appendix D.3). Of note, though our framework will be formulated for policy-based methods, our final form of cost function (Equation (6.9) in Section 6.4) is applicable to both policy-based and value-based methods. We propose a novel k-*shortest-path* (k-SP) constraint (Definition 8) that improves sample efficiency of policy learning (See Figure 6.1). The k-SP constraint is applied to a trajectory rolled out by a policy; all of its sub-path of length k is required to be a shortest-path under the π-*distance* metric which we define in Section 6.3.1. We prove that applying our constraint preserves the optimality for any MDP (Theorem 10), except the stochastic and multi-goal MDP which requires additional assumptions. We relax the hard constraint into a soft cost formulation [Tessler et al., 2019], and use a *reachability network* [Savinov et al., 2018b] (RNet) to efficiently learn the cost function in an off-policy manner.

We summarize our contributions as the following: **(1)** We propose a novel constraint that can improve the sample efficiency of any model-free RL method in sparse reward tasks. **(2)** We present several theoretical results including the proof that our proposed constraint preserves the optimal policy of given MDP. **(3)** We present a numerical result in tabular RL setting to precisely evaluate the effectiveness of the proposed method. **(4)** We propose a practical way to implement our proposed constraint, and demonstrate that it provides a significant improvement on three complex deep RL domains. **(5)** We demonstrate that our method significantly improves the sample-efficiency of PPO, and outperforms existing novelty-seeking methods on three complex domains in sparse reward setting.

6.2 Related Work

Shortest-path Problem and Planning. Many early works [Bellman, 1958, Ford Jr, 1956, Bertsekas and Tsitsiklis, 1991, 1995] have discussed (stochastic) shortest path problems in the context of MDP. They viewed the shortest-path problem as planning problem and proposed a dynamic programming-based algorithm similar to the value iteration [Sutton and Barto, 2018] to solve it. Our main idea is inspired by (but not based on) this viewpoint. Specifically, our method does not directly solve the shortest path problem via planning; hence, our method does not require model. Our method only exploits the optimality guarantee of the shortest-path under the π-distance to prune out sub-optimal policies (*i.e.*, non-shortest paths).

Distance Metric in Goal-conditioned RL. In goal-conditioned RL, there has been a recent surge of interest on learning a distance metric in state (or goal) space for to construct a high-level MDP graph and perform planning to find a shortest-path to the goal state. Huang et al. [2019], Laskin et al. [2020] used the universal value function (UVF) [Schaul et al., 2015] with a constant step

penalty as a distance function. Zhang et al. [2018], Laskin et al. [2020] used the success rate of transition between nodes as distance and searched for the longest path to find the plan with highest success rate. SPTM [Savinov et al., 2018a] defined a binary distance based on reachability network (RNet) to connect near by nodes in the graph. However, the proposed distance metrics and methods can be used only for the goal-conditioned task and lacks the theoretical guarantee in general MDP, while our theory and framework are applicable to general MDP (see Section 6.3.1).

Reachability Network. The reachability network (RNet) was first proposed by Savinov et al. [2018b] as a way to measure the novelty of a state for *exploration*. Intuitively, if current state is not reachable from previous states in episodic memory, it is considered to be novel. SPTM [Savinov et al., 2018a] used RNet to predict the local connectivity (*i.e.*, binary distance) between observations in memory for *graph-based planning* in navigation task. On the other hand, we use RNet for *constraining the policy* (*i.e.*, removing the sub-optimal policies from policy space). Thus, in ours and in other two compared works, RNet is being employed for fundamentally different purposes.

Approximate state abstraction. The approximate state abstraction approaches investigate partitioning an MDP's state space into clusters of similar states while preserving the optimal solution. Researchers have proposed several state similarity metrics for MDPs. Dean et al. [2013] proposed to use the bisimulation metrics [Givan et al., 2003, Ferns et al., 2004], which measures the difference in transition and reward function. Bertsekas et al. [1988] used the magnitude of Bellman residual as a metric. Abel et al. [2016, 2018], Li et al. [2006] used the different types of distance in optimal Q-value to measure the similarity between states to bound the sub-optimality in optimal value after the abstraction. Recently, Castro [2019] extended the bisimulation metrics to the approximate version for deep-RL setting where tabular representation of state is not available. Our shortest-path constraint can be seen as a form of state abstraction, in that ours also aim to reduce the size of MDP (*i.e.*, state and action space) while preserving the "solution quality". However, our method does so by removing sub-optimal policies, not by aggregating similar states (or policies).

6.3 Formulation: k-shortest-path Constraint

We define the k-shortest-path (k-SP) constraint to remove redundant transitions (*e.g.*, unnecessarily going back and forth), leading to faster policy learning. We show two important properties of our constraint: (1) the optimal policy is preserved, and (2) the policy search space is reduced.

In this work, we limit our focus to MDPs satisfying $R(s) + \gamma V^*(s) > 0$ for all initial states $s \in \rho$ and all rewarding states that optimal policy visits with non-zero probability $s \in \{s | r(s) \neq 0, \pi^*(s) > 0\}$. We exploit this mild assumption to prove that our constraint preserves optimality.

Intuitively, we exclude the case when the optimal strategy for the agent is at best choosing a "lesser of evils" (*i.e.*, largest but negative value) which often still means a failure. We note that this is often caused by unnatural reward function design; in principle, we can avoid this by simply offsetting reward function by a constant $-|\min_{s\in\{s|\pi^*(s)>0\}} V^*(s)|$ for every transition, assuming the policy is *proper*[1]. Goal-conditioned RL [Nachum et al., 2018] and most of the well-known domains such as *Atari* [Bellemare et al., 2013], *DeepMind Lab* [Beattie et al., 2016], *MiniGrid* [Chevalier-Boisvert et al., 2018], etc., satisfy this assumption. Also, for general settings with stochastic MDP and multi-goals, we require additional assumptions to prove the optimality guarantee (See Appendix D.3 for details).

6.3.1 Shortest-path Policy and Shortest-path Constraint

Let τ be a *path* defined by a sequence of states: $\tau = \{s_0, \dots, s_{\ell(\tau)}\}$, where $\ell(\tau)$ is the *length* of a path τ (*i.e.*, $\ell(\tau) = |\tau| - 1$). We denote the set of all paths from s to s' by $\mathcal{T}_{s,s'}$. A path τ^* from s to s' is called a *shortest path* from s to s' if $\ell(\tau^*)$ is minimum, *i.e.*, $\ell(\tau^*) = \min_{\tau\in\mathcal{T}_{s,s'}} \ell(\tau)$.

Now we will define similar concepts (length, shortest path, *etc.*) *with respect to* a policy. Intuitively, a policy that rolls out shortest paths (up to some stochasticity) to a goal state or between any state pairs should be a counterpart. We consider a set of all admissible paths from s to s' under a policy π:

Definition 1 (Path set)**.** $\mathcal{T}^\pi_{s,s'} = \{\tau \mid s_0 = s, s_{\ell(\tau)} = s', p_\pi(\tau) > 0, s_t \neq s' \text{ for } \forall t < \ell(\tau)\}$. That is, $\mathcal{T}^\pi_{s,s'}$ is a set of all paths that policy π may roll out from s and terminate once visiting s'.

If the MDP is a single-goal task, *i.e.*, there exists a unique (rewarding) goal state $s_g \in \mathcal{S}$ such that s_g is a terminal state, and $R(s) > 0$ if and only if $s = s_g$, any shortest path from an initial state to the goal state is the optimal path with the highest return $R(\tau)$, and a policy that rolls out a shortest path is therefore optimal (see Lemma 11).[2] This is because all states except for s_g are non-rewarding states, but in general MDPs this is not necessarily true. However, this motivates us to limit the domain of shortest path to among *non-rewarding states*. We define *non-rewarding paths* from s to s' as follows:

Definition 2 (Non-rewarding path set)**.** $\mathcal{T}^\pi_{s,s',\mathrm{nr}} = \{\tau \mid \tau \in \mathcal{T}^\pi_{s,s'}, r_t = 0 \text{ for } \forall t < \ell(\tau)\}$.

In words, $\mathcal{T}^\pi_{s,s',\mathrm{nr}}$ is a set of all non-rewarding paths from s to s' rolled out by policy π (*i.e.*, $\tau \in \mathcal{T}^\pi_{s,s'}$) without any associated reward except the last step (*i.e.*, $r_t = 0$ for $\forall t < \ell(\tau)$). Now we are ready to define a notion of length with respect to a policy and shortest path policy:

[1] It is an instance of potential-based reward shaping which has optimality guarantee [Ng et al., 1999].

[2] We refer the readers to Appendix D.2 for more detailed discussion and proofs for single-goal MDPs.

Definition 3 (π-distance from s to s')**.** $D^{\pi}_{\mathrm{nr}}(s, s') = \log_{\gamma}\left(\mathbb{E}_{\tau\sim\pi:\ \tau\in\mathcal{T}^{\pi}_{s,s',\mathrm{nr}}}\left[\gamma^{\ell(\tau)}\right]\right)$

Definition 4 (Shortest path distance from s to s')**.** $D_{\mathrm{nr}}(s, s') = \min_{\pi} D^{\pi}_{\mathrm{nr}}(s, s')$.

We define π-distance to be the log-mean-exponential of the length $\ell(\tau)$ of non-rewarding paths $\tau \in \mathcal{T}^{\pi}_{s,s',\mathrm{nr}}$. When there exists no admissible path from s to s' under policy π, the path length is defined to be ∞: $D^{\pi}_{\mathrm{nr}}(s, s') = \infty$ if $\mathcal{T}^{\pi}_{s,s',\mathrm{nr}} = \emptyset$. We note that when both MDP and policy are deterministic, $D^{\pi}(s, s')$ recovers the natural definition of path length, $D^{\pi}_{\mathrm{nr}}(s, s') = \ell(\tau)$.

We call a policy a *shortest-path policy* from s to s' if it roll outs a path with the smallest π-distance:

Definition 5 (Shortest path policy from s to s')**.** $\pi \in \Pi^{\mathrm{SP}}_{s\to s'} = \{\pi \in \Pi \mid D^{\pi}_{\mathrm{nr}}(s, s') = D_{\mathrm{nr}}(s, s')\}$.

Finally, we will define the shortest-path (SP) constraint. Let $\mathcal{S}^{\mathrm{IR}} = \{s \mid R(s) > 0 \text{ or } \rho(s) > 0\}$ be the union of all initial and rewarding states, and $\Phi^{\pi} = \{(s, s') \mid s, s' \in \mathcal{S}^{\mathrm{IR}}, \rho(s) > 0, \mathcal{T}^{\pi}_{s,s',\mathrm{nr}} \neq \emptyset\}$ be the subset of $\mathcal{S}^{\mathrm{IR}}$ such that agent may roll out. Then, the SP constraint is applied to the non-rewarding sub-paths between states in Φ^{π}: $\mathcal{T}^{\pi}_{\Phi,\mathrm{nr}} = \bigcup_{(s,s')\in\Phi^{\pi}} \mathcal{T}^{\pi}_{s,s',\mathrm{nr}}$. We note that these definitions are used in the proofs (Appendix D.3). Now, we define the shortest-path constraint as follows:

Definition 6 (Shortest-path constraint)**.** A policy π satisfies the shortest-path (SP) constraint if $\pi \in \Pi^{\mathrm{SP}}$, where $\Pi^{\mathrm{SP}} = \{\pi \mid \text{For all } s, s' \in \mathcal{T}^{\pi}_{\Phi,\mathrm{nr}}, \text{it holds } \pi \in \Pi^{\mathrm{SP}}_{s\to s'}\}$.

Intuitively, the SP constraint forces a policy to transition between initial and rewarding states via shortest paths. The SP constraint would be particularly effective in sparse-reward settings, where the distance between rewarding states is large.

Given these definitions, we can show that an optimal policy indeed satisfies the SP constraint in a general MDP setting. In other words, the shortest path constraint should not change optimality:

Theorem 7. *For any MDP, an optimal policy π^* satisfies the shortest-path constraint: $\pi^* \in \Pi^{SP}$.*

Proof. See Appendix D.3 for the proof. □

6.3.2 Relaxation: k-shortest-path Constraint

Implementing the shortest-path constraint is, however, intractable since it requires a distance predictor $D_{\mathrm{nr}}(s, s')$. Note that the distance predictor addresses the optimization problem, which might be as difficult as solving the given task. To circumvent this challenge, we consider its more tractable version, namely a k-*shortest path* constraint, which reduces the shortest-path problem

$D_{\text{nr}}(s, s')$ to a binary decision problem — is the state s' reachable from s within k steps? — also known as k-reachability [Savinov et al., 2018b]. The k-shortest path constraint is defined as follows:

Definition 8 (k-shortest-path constraint)**.** A policy π satisfies the k-shortest-path constraint if $\pi \in \Pi_k^{\text{SP}}$, where

$$\Pi_k^{\text{SP}} = \{\pi \mid \text{For all } s, s' \in \mathcal{T}_{\Phi,\text{nr}}^{\pi}, D_{\text{nr}}^{\pi}(s, s') \leq k, \\ \text{it holds } \pi \in \Pi_{s \to s'}^{\text{SP}}\}. \tag{6.1}$$

Note that the SP constraint (Definition 6) is relaxed by adding a condition $D_{\text{nr}}^{\pi}(s, s') \leq k$. In other words, the k-SP constraint is imposed only for s, s'-path whose length is not greater than k. From Equation (6.1), we can prove an important property and then Theorem 10 (optimality):

Lemma 9. *For an MDP $\mathcal{M}$, $\Pi_m^{SP} \subset \Pi_k^{SP}$ if $k < m$.*

Proof. It is true since $\{(s, s') \mid D_{\text{nr}}^{\pi}(s, s') \leq k\} \subset \{(s, s') \mid D_{\text{nr}}^{\pi}(s, s') \leq m\}$ for $k < m$. □

Theorem 10. *For an MDP $\mathcal{M}$ and any $k \in \mathbb{R}$, an optimal policy π^* is a k-shortest-path policy.*

Proof. Theorem 7 tells $\pi^* \in \Pi^{\text{SP}}$. Equation (6.1) tells $\Pi^{\text{SP}} = \Pi_{\infty}^{\text{SP}}$ and Lemma 9 tells $\Pi_{\infty}^{\text{SP}} \subset \Pi_k^{\text{SP}}$. Collectively, we have $\pi^* \in \Pi^{\text{SP}} = \Pi_{\infty}^{\text{SP}} \subset \Pi_k^{\text{SP}}$. □

In conclusion, Theorem 10 states that the k-SP constraint does not change the optimality of policy, and Lemma 9 states a larger k results in a larger reduction in policy search space. Thus, it motivates us to apply the k-SP constraint in policy search to more efficiently find an optimal policy. For the numerical experiment on measuring the reduction in the policy roll-outs space, please refer to Section 6.5.5.

6.4 SPRL: Shortest-Path Reinforcement Learning

k-Shortest-Path Cost. The objective of RL with the k-SP constraint Π_k^{SP} can be written as:

$$\pi^* = \arg\max_{\pi} \mathbb{E}^{\pi}\left[R(\tau)\right], \quad \text{s.t. } \pi \in \Pi_k^{\text{SP}}, \tag{6.2}$$

where $\Pi_k^{\text{SP}} = \{\pi \mid \forall (s, s' \in \mathcal{T}_{\Phi,\text{nr}}^{\pi}), D_{\text{nr}}^{\pi}(s, s') \leq k, \text{it holds } \pi \in \Pi_{s \to s'}^{\text{SP}}\}$ (Definition 8). We want to formulate the constraint $\pi \in \Pi_k^{\text{SP}}$ in the form of constrained MDP (Section 2.1), *i.e.*, as $C(\pi) \leq \alpha$.

We begin by re-writing the k-SP constraint into a cost-based form:

$$\Pi_k^{\text{SP}} = \{\pi \mid C_k^{\text{SP}}(\pi) = 0\}, \text{ where } C_k^{\text{SP}}(\pi) = \sum_{(s,s' \in \mathcal{T}_{\Phi,\text{nr}}^{\pi}): D_{\text{nr}}^{\pi}(s,s') \leq k} \mathbb{I}\left[D_{\text{nr}}(s,s') < D_{\text{nr}}^{\pi}(s,s')\right]. \quad (6.3)$$

Note that $\mathbb{I}\left[D_{\text{nr}}(s,s') < D_{\text{nr}}^{\pi}(s,s')\right] = 0 \leftrightarrow D_{\text{nr}}(s,s') = D_{\text{nr}}^{\pi}(s,s')$ since $D_{\text{nr}}(s,s') \leq D_{\text{nr}}^{\pi}(s,s')$ from Definition 4. Similar to Tessler et al. [2019], we apply the constraint to the on-policy trajectory $\tau = (s_0, s_1, \ldots)$ with discounting by replacing (s,s') with (s_t, s_{t+l}) where $[t, t+l]$ represents each segment of τ with length l:

$$\begin{aligned}
C_k^{\text{SP}}(\pi) &\simeq \mathbb{E}_{\tau\sim\pi}\left[\textstyle\sum_{(t,l):t\geq 0, l\leq k} \gamma^t \cdot \mathbb{I}\left[D_{\text{nr}}(s_t, s_{t+l}) < D_{\text{nr}}^{\pi}(s_t, s_{t+l})\right] \cdot \mathbb{I}\left[\{r_j\}_{j=t}^{t+l-1} = 0\right]\right] \\
&= \mathbb{E}_{\tau\sim\pi}\left[\textstyle\sum_{(t,l):t\geq 0, l\leq k} \gamma^t \cdot \mathbb{I}\left[D_{\text{nr}}(s_t, s_{t+l}) < \log_\gamma\left(\mathbb{E}_{\tau\in\mathcal{T}_{s_t,s_{t+l},\text{nr}}^{\pi}}\left[\gamma^{|\tau|}\right]\right)\right] \cdot \mathbb{I}\left[\{r_j\}_{j=t}^{t+l-1} = 0\right]\right] \\
&\leq \mathbb{E}_{\tau\sim\pi}\left[\textstyle\sum_{(t,l):t\geq 0, l\leq k} \gamma^t \cdot \mathbb{I}\left[D_{\text{nr}}(s_t, s_{t+l}) < k\right] \cdot \mathbb{I}\left[\{r_j\}_{j=t}^{t+l-1} = 0\right]\right] \triangleq \hat{C}_k^{\text{SP}}(\pi). \quad (6.4)
\end{aligned}$$

Note that it is sufficient to consider only the cases $l = k$ (because for $l < k$, given $D_{\text{nr}}(s_t, s_{t+k}) < k$, we have $D(s_t, s_{t+l}) \leq l < k$). Then, we simplify $\hat{C}_k^{\text{SP}}(\pi)$ as

$$\begin{aligned}
\hat{C}_k^{\text{SP}}(\pi) &= \mathbb{E}_{\tau\sim\pi}\left[\textstyle\sum_t \gamma^t \mathbb{I}\left[D_{\text{nr}}(s_t, s_{t+k}) < k\right] \cdot \mathbb{I}\left[\{r_j\}_{j=t}^{t+k-1} = 0\right]\right] \quad (6.5) \\
&= \mathbb{E}_{\tau\sim\pi}\left[\textstyle\sum_t \gamma^t \mathbb{I}[t \geq k] \cdot \mathbb{I}\left[D_{\text{nr}}(s_{t-k}, s_t) < k\right] \cdot \mathbb{I}\left[\{r_j\}_{j=t-k}^{t-1} = 0\right]\right]. \quad (6.6)
\end{aligned}$$

Finally, the per-time step cost c_t is given as:

$$c_t = \mathbb{I}[t \geq k] \cdot \mathbb{I}\left[D_{\text{nr}}(s_{t-k}, s_t) < k\right] \cdot \mathbb{I}\left[\{r_j\}_{j=t-k}^{t-1} = 0\right], \quad (6.7)$$

where $\hat{C}_k^{\text{SP}}(\pi) = \mathbb{E}_{\tau\sim\pi}\left[\sum_t \gamma^t c_t\right]$. Note that $\hat{C}_k^{\text{SP}}(\pi)$ is an upper bound of $C_k^{\text{SP}}(\pi)$, which will be minimized by the bound to make as little violation of the shortest-path constraint as possible. Intuitively speaking, c_t penalizes the agent from taking a non-k-shortest path at each step, so minimizing such penalties will make the policy satisfy the k-shortest-path constraint. In Equation (6.7), c_t depends on the previous k steps; hence, the resulting CMDP becomes a $(k+1)$-th order MDP. In practice, however, we empirically found that feeding only the current time-step observation to the policy performs better than stacking the previous k-steps of observations. Thus, we did not stack the observation in all the experiments. We use the Lagrange multiplier method to convert the objective (6.2) into an equivalent unconstrained problem as follows:

$$\min_{\lambda>0} \max_{\theta} L(\lambda, \theta) = \min_{\lambda>0} \max_{\theta} \mathbb{E}_{\tau\sim\pi_\theta}\Big[\textstyle\sum_t \gamma^t \left(r_t - \lambda c_t\right)\Big], \quad (6.8)$$

Require: Hyperparameters: $k \in \mathbb{N}, \lambda > 0$
for $n = 1, \ldots, N_{\text{policy}}$ **do**
  Rollout transitions $\tau = \{s_t, a_t, r_t\}_{t=1}^{|\tau|} \sim \pi$
  Compute the cost term $\{c_t\}_{t=1}^{|\tau|}$ as Equation (6.12).
  Update policy π to maximize the objective as Equation (6.9) (e.g., run PPO train steps).
  Update Rnet training-buffer $\mathcal{B} = \mathcal{B} \cup \{\tau\}$.
  if $n \% T_{\text{Rnet}} = 0$ **then** ▷ Periodically train Rnet for N_{Rnet} times
    for $m = 1, \ldots, N_{\text{Rnet}}$ **do**
      Sample triplet $(s_{\text{anc}}, s_+, s_-) \sim \mathcal{B}$.
      Update Rnet to minimize $\mathcal{L}_{\text{Rnet}}$ as Equation (6.13).
    end for
  end if
end for

Program 6.1: Reinforcement Learning with k-SP constraint (SPRL)

where L is the Lagrangian, θ is the parameter of policy π, and $\lambda > 0$ is the Lagrangian multiplier. Since Theorem 10 shows that the shortest-path constraint preserves the optimality, we are free to set any $\lambda > 0$. Thus, we simply consider λ as a tunable *positive* hyperparameter, and simplify the min-max problem (6.8) to an RL objective with costs c_t being added:

$$\max_{\theta} \mathbb{E}_{\tau \sim \pi_\theta} \Big[\textstyle\sum_t \gamma^t \left(r_t - \lambda c_t \right) \Big]. \tag{6.9}$$

Practical implementation of the cost function. We implement the binary distance discriminator $\mathbb{I}(D_{\text{nr}}(s_{t-k}, s_t) < k)$ in Equation (6.7) using *k-reachability network* [Savinov et al., 2018b]. The k-reachability network $\text{Rnet}_k(s, s')$ is trained to output 1 if the state s' is reachable from the state s with less than or equal to k consecutive actions, and 0 otherwise. Formally, we take the functional form: $\text{Rnet}_k(s, s') \simeq \mathbb{I}\left(D_{\text{nr}}(s, s') < k + 1\right)$. We then estimate the cost term c_t using $(k-1)$-reachability network as follows:

$$\begin{aligned} c_t &= \mathbb{I}\left[D_{\text{nr}}(s_{t-k}, s_t) < k\right] \cdot \mathbb{I}\left[\{r_l\}_{l=t-k}^{t-1} = 0\right] \cdot \mathbb{I}\left[t \geq k\right] && (6.10) \\ &= \text{Rnet}_{k-1}(s_{t-k}, s_t) \cdot \mathbb{I}\left[\{r_l\}_{l=t-k}^{t-1} = 0\right] \cdot \mathbb{I}\left[t \geq k\right]. && (6.11) \end{aligned}$$

Intuitively speaking, if the agent takes a k-shortest path, then the distance between s_{t-k} and s_t is k, hence $c_t = 0$. If it is not a k-shortest path, $c_t > 0$ since the distance between s_{t-k} and s_t will be less than k. In practice, due to the error in the reachability network, we add a small tolerance $\Delta t \in \mathbb{N}$ to

ignore outliers. It leads to an empirical version of the cost as follows:

$$c_t \simeq \text{Rnet}_{k-1}(s_{t-k-\Delta t}, s_t) \cdot \mathbb{I}\left[\{r_l\}_{l=t-k-\Delta t}^{t-1} = 0\right] \cdot \mathbb{I}(t \geq k + \Delta t). \tag{6.12}$$

In our experiment, we found that a small tolerance $\Delta t \simeq k/5$ works well in general. Similar to Savinov et al. [2018b], we used the following contrastive loss for training the reachability network:

$$\mathcal{L}_{\text{Rnet}} = -\log\left(\text{Rnet}_{k-1}(s_{\text{anc}}, s_+)\right) - \log\left(1 - \text{Rnet}_{k-1}(s_{\text{anc}}, s_-)\right), \tag{6.13}$$

where s_{anc}, s_+, s_- are the anchor, positive, and negative samples, respectively.

6.5 Experiments

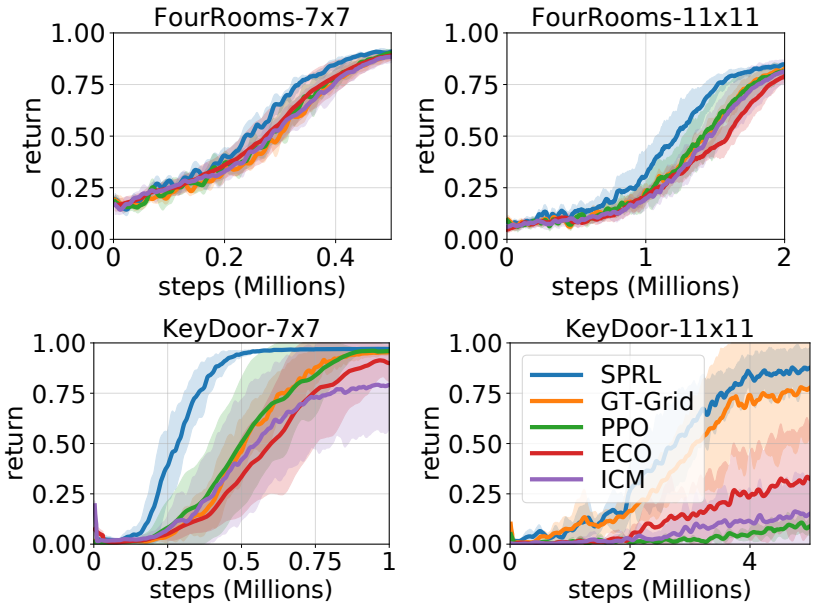


Figure 6.2: Progress of average episode reward on *MiniGrid* tasks. We report the mean (solid curve) and standard error (shadowed area) of the performance over six random seeds.

6.5.1 Settings

Environments. We evaluate our **SPRL** on three challenging domains: *MiniGrid* [Chevalier-Boisvert et al., 2018], *DeepMind Lab* [Beattie et al., 2016] and *Atari* [Bellemare et al., 2013]. *MiniGrid* is a 2D grid world environment with challenging features such as pictorial observation, random initialization of the agent and the goal, complex state and action space where coordinates,

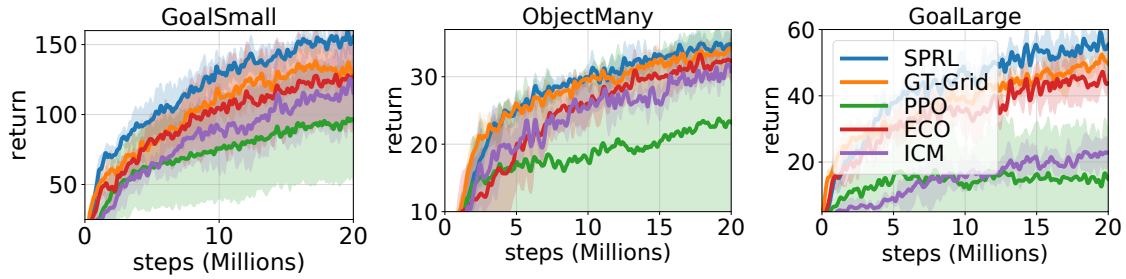


Figure 6.3: Progress of average episode reward on *DeepMind Lab* tasks. We report the mean (solid curve) and standard error (shadowed area) of the performance over four random seeds.

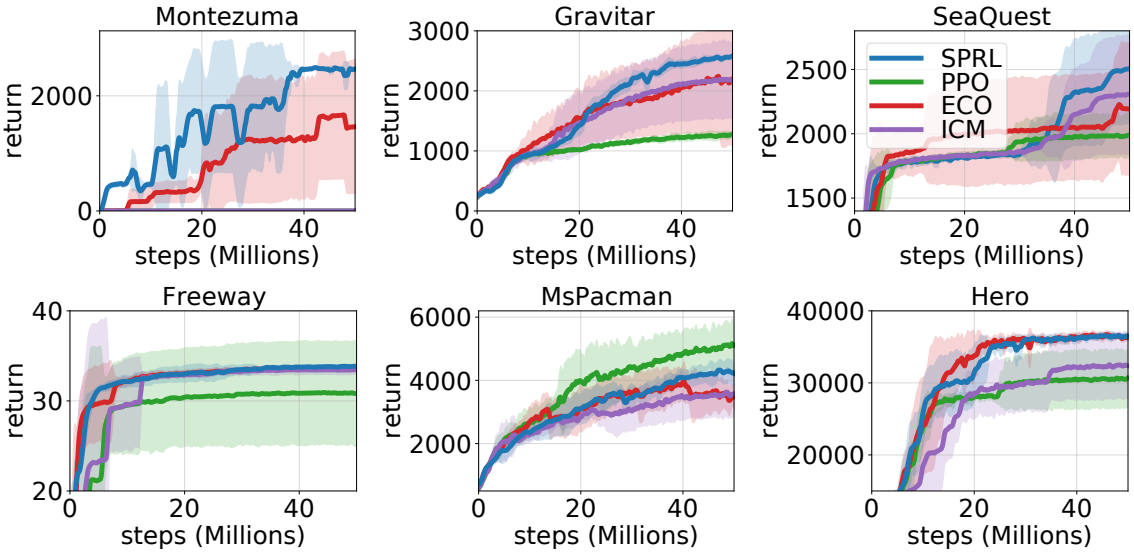


Figure 6.4: Progress of average episode reward on *Atari* tasks. We report the mean (solid curve) and standard error (shadowed area) of the performance over four random seeds.

directions, and other object statuses (*e.g.*, key-door) are considered. We conducted experiments on four standard tasks: *FourRooms-7×7*, *FourRooms-11×11*, *KeyDoors-7×7*, and *KeyDoors-11×11*. *DeepMind Lab* is a 3D environment with first person view. Along with the nature of partially-observed MDP, at each episode, the agent's initial and the goal location are reset randomly with a change of texture, maze structure, and colors. We conducted experiments on three standard tasks: *GoalSmall*, *GoalLarge*[3] , and *ObjectMany*. For *Atari*, among 52 games we chose two sparse-reward tasks (*Montezuma's Revenge*, *Freeway*), one dense-reward task (*Ms.Pacman*), and three non-navigational tasks (*Gravitar*, *Seaquest*, *HERO*) where the agent receives reward by hitting the enemy by firing a bullet or removing the obstacle by installing a bomb. See Appendix D.4, Appendix D.5, Appendix D.6 for more details of *MiniGrid*, *DeepMind Lab*, and *Atari* respectively.

[3] *GoalLarge* task corresponds to the *Sparse* task in Savinov et al. [2018b], and our Figure 6.3 reproduces the result reported.

Baselines. We compared our methods with four baselines: **PPO** [Schulman et al., 2017], *episodic curiosity* (**ECO**) [Savinov et al., 2018b], *intrinsic curiosity module* (**ICM**) [Pathak et al., 2017], and **GT-Grid** [Savinov et al., 2018b]. The **PPO** is used as a baseline RL algorithm for all other agents. The **ECO** agent is rewarded when it visits a state that is not reachable from the states in episodic memory within a certain number of actions; thus the novelty is only measured within an episode. Following Savinov et al. [2018b], we trained RNet in an off-policy manner from the agent's experience and used it for our **SPRL** and **ECO** on *MiniGrid* (Section 6.5.2), *DeepMind Lab* (Section 6.5.3) and *Atari* (Section 6.5.4). The **GT-Grid** agent has access to the agent's (x, y) coordinates. It uniformly divides the world in 2D grid cells, and the agent is rewarded for visiting a novel grid cell. The **ICM** agent learns a forward and inverse dynamics model and uses the prediction error of the forward model to measure the novelty. We used the publicly available codebase [Savinov et al., 2018b] to obtain the baseline results. We used the same hyperparameter for all the tasks for a given domain — the details are described in the Appendix. We used the standard domain and tasks for reproducibility.

6.5.2 Results on *MiniGrid*

Figure 6.2 shows the performance of all the methods on *MiniGrid* domain. **SPRL** consistently outperforms all the baseline methods over all tasks. We observe that exploration-based methods (*i.e.*, **ECO**, **ICM**, and **GT-Grid**) perform similarly to the **PPO** in the tasks with small state space (*e.g.*, *FourRooms-7×7* and *KeyDoors-7×7*). However, **SPRL** demonstrates a significant performance gain since it improves the exploitation by avoiding sub-optimality caused by taking a non-shortest-path.

6.5.3 Results on *DeepMind Lab*

Figure 6.3 shows the performance of all the methods on *DeepMind Lab* tasks. Overall, **SPRL** achieves superior results compared to other methods. By the task design, the difficulty of exploration increases in the order of *GoalSmall*, *ObjectMany*, and *GoalLarge* tasks, and we observe a coherent trend in the result. For harder exploration tasks, the exploration-based methods (**GT-Grid**, **ICM** and **ECO**) achieve a larger improvement over **PPO**: *e.g.*, 20%, 50%, and 100% improvement in *GoalSmall*, *ObjectMany*, and *GoalLarge*, respectively. As shown in Lemma 9, **SPRL** is expected to have larger improvement for larger trajectory space and sparser reward settings. We can verify this from the result: **SPRL** has the largest improvement in *GoalLarge* task, where both the map is largest and the reward is most sparse. Interestingly, **SPRL** even outperforms **GT-Grid** which simulates the upper-bound performance of novelty-seeking exploration method. This is possible since **SPRL** improves the exploration by suppressing unnecessary explorations, which is different from novelty-seeking methods, and also improves the exploitation by reducing the policy search

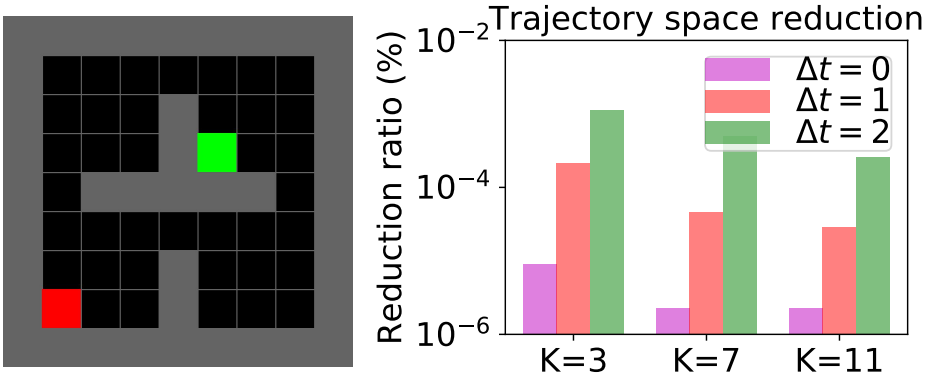


Figure 6.5: (Left) 7×7 Tabular four-rooms domain with initial agent location (red) and the goal location (green). (Right) The trajectory space reduction ratio (%) before and after constraining the trajectory space for various k and Δt with k-SP constraint. Even a small k can greatly reduce the trajectory space with a reasonable tolerance Δt.

space.

(a) **Random** (b) **GT-UCB** (c) **SPRL** (d) **SPRL**+Reward

Figure 6.6: Transition count maps for baselines and **SPRL**: (a), (b), and (c) are in *reward-free* while (d) is in *reward-aware* setting. In reward-free settings (a-c), we show rewarding states in light green only for visualization purpose, but the agent does not receive rewards from the environment. The location of agent's initial state (orange) and rewarding states (dark green) are fixed. The episode length is limited to 500 steps.

6.5.4 Results on *Atari*

One of the main challenges in *Atari* is the distribution shift in the state space within a task. Unlike *MiniGrid* and *DeepMind Lab*, many *Atari* tasks involve the transition between different rooms in each of which the agent observes significantly different set of states. This induces an instability in the RNet training; RNet often overfits to the initial room and performs poorly when the agent navigates to the different rooms. To mitigate this problem, we added the weight decay for the RNet training. For other technical details, please refer to Appendix D.6.3.

Figure 6.4 summarizes the performance of all the methods on *Atari* tasks. **SPRL** outperforms all the baseline methods on five out of six tasks except for *Ms.Pacman*, which is a dense reward task. We note that other exploration methods, **ICM** and **ECO**, also perform poor on this task. For the sparse reward task, especially in *Montezuma's Revenge*, **SPRL** achieves the performance comparable to the SOTA exploration methods such as RND [Burda et al., 2018] (1000 score at 50M steps with 32 parallel environments. **SPRL** used 12 parallel environments.) and SOTA exploitation methods such as SIL [Oh et al., 2018] (2500 score at 50M steps). Lastly, **SPRL** achieves the largest improvement to the **PPO** in non-navigational tasks (*Gravitar*, *Seaquest*, *HERO*). This verifies that our k-SP constraint is not limited to just the geometric path but can be applied to any general trajectory, or a sequence of state transitions, in MDP.

6.5.5 Quantitative Analysis on k-SP Constraint

In this section, we numerically evaluate the effect of our k-shortest path constraint in tabular-RL setting. Specifically, we study the following questions: (1) Does the k-SP constraint with larger k results in more reduction in trajectory space? (*i.e.*, validation of Lemma 9) (2) How much reduction in trajectory space does k-SP constraint provide with different k and tolerance Δt?

We implemented a simple tabular 7x7 four-rooms domain where each state maps to a unique (x, y) location of the agent. The agent can take *up, down, left, right* primitive actions to move to the neighboring state, and the episode horizon is set to 14 steps. The goal of the agent is reaching to the goal state, which gives +1 reward and terminates the episode. We computed the ground-truth distance between a pair of states to implement the k-shortest path constraint. We used the ground-truth distance function instead of the learned RNet to implement the exact SPRL agent.

Figure 6.5 summarizes the reduction in the trajectory space size. We searched over all possible trajectories of length 14 using breadth-first-search (BFS). Then we counted the number of trajectories satisfying our k-SP constraint with varying parameters k and tolerance Δt and divided by total number of trajectories (*i.e.*, $4^{14} = 268$M). The result shows that our k-SP constraint drastically reduces the trajectory space even in a simple 2D grid domain; with very small $k = 3$ and no tolerance $\Delta t = 2$, we get only $24/268$M size of the original search space. As we increase k, we can see more reduction in the trajectory space, which is consistent with Lemma 9. Also, increasing the tolerance Δt slightly hurts the performance, but still achieves a large reduction

6.5.6 Qualitative Analysis on *MiniGrid*

We qualitatively studied what type of policy is learned with the k-SP constraint with the ground-truth RNet in *NineRooms* domain of *MiniGrid*. Figure 6.6 (a-c) shows the converged behavior of **SPRL** ($k = 15$), the ground-truth count-based exploration [Lai and Robbins, 1985] agent (**GT-UCB**) and

uniformly random policy (**Random**) in a reward-free setting. We counted all the state transitions ($s_t \rightarrow s_{t+1}$) of each agent's roll-out and averaged over 4 random seeds. **Random** cannot explore further than the initial few rooms. **GT-UCB** seeks for a novel states, and visits all the states uniformly. **SPRL** learns to take a longest possible shortest path, which results in a "straight" path across the rooms. Note that this only represents a partial behavior of **SPRL**, since our cost also considers the *existence of non-zero reward* (see Equation (6.7)). Thus, in (d), we tested **SPRL** while providing only the *existence* of non-zero reward (but not the reward magnitude). **SPRL** learns to take a shortest path between rewarding and initial states that is consistent with the shortest-path definition in Definition 8.

6.6 Discussion

We presented the k-shortest-path constraint, which can improve the sample-efficiency of any model-free RL method by preventing the agent from taking sub-optimal transitions. We empirically showed that **SPRL** outperforms vanilla RL and strong novelty-seeking exploration baselines on three challenging domains. We believe that our framework develops a unique direction for improving the sample efficiency in reinforcement learning; hence, combining our work with other techniques for better sample efficiency will be an interesting future work that could benefit many practical tasks.

CHAPTER 7

Learning Factored Task Structure for Generalization to Unseen Entities

Real world tasks are hierarchical and compositional. Solving these tasks efficiently requires long horizon planning and reasoning. We propose to tackle this problem in a few-shot RL setting – in an adaptation phase, an agent must first explore the environment to infer the latent hierarchical and compositional structure. Then in a test phase, the agent uses this latent information to maximize reward in the test environment. We formulate *predicate subtask graph inference* (PSGI), a method for inferring the latent *predicate subtask graph* of the environment, which models preconditions and dependencies of subtasks in a first order logic manner. We infer subtasks with *predicate* (symbolic) form (e.g `pickup` X) as nodes in a graph structure. To facilitate this, we learn *parameter attributes* in a zero-shot manner, which are used to differentiate the structure of predicate subtasks (e.g. $\hat{f}_{\text{pickable}}(X)$). We show this approach accurately learns the latent structure on hierarchical and compositional tasks more efficiently than prior work, and show PSGI can generalize by modelling structure on subtasks *unseen* during adaptation.

7.1 Introduction

Real world tasks are *hierarchical*. Hierarchical tasks are composed of multiple sub-goals that must be completed in certain order. For example, the cooking task shown in Figure 7.1 requires an agent to boil some food object (e.g. `Cooked egg`). An agent must place the food object x in a cookware object y, place the cookware object on the stove, before boiling this food object x. Parts of this task can be decomposed into sub-goals, or *subtasks* (e.g. `Pickup egg`, `Put egg on pot`). Solving these tasks requires long horizon planning and reasoning ability Erol [1996], Xu et al. [2017], Ghazanfari and Taylor [2017], Sohn et al. [2018]. This problem is made more difficult of rewards are *sparse*, if only few of the subtasks in the environment provide reward to the agent.

Real world tasks are also *compositional* Carvalho et al. [2020], Loula et al. [2018], Andreas et al. [2017], Oh et al. [2017]. Compositional tasks are often made of different "components" that can

recombined to form new tasks. These components can be numerous, leading to a *combinatorial* number of subtasks. For example, the cooking task shown in Figure 7.1 contains subtasks that follow a verb-objects structure. The verb `Pickup` admits many subtasks, where any object x composes into a new subtask (e.g. `Pickup egg`, `Pickup pot`). Solving compositional tasks also requires reasoning Andreas et al. [2017], Oh et al. [2017]. Without reasoning on the relations between components between tasks, exploring the space of a combinatorial number of subtasks is extremely inefficient.

In this work, we propose to tackle the problem of hierarchical *and* compositional tasks. Prior work has tackled learning hierarchical task structures by modelling dependencies between subtasks in a *graph structure* Sohn et al. [2018, 2020], Xu et al. [2017], Huang et al. [2018]. In these settings, during training, the agent tries to efficiently adapt to a task by inferring the latent graph structure, then uses the inferred graph to maximize reward during test. However, this approach does not scale for compositional tasks. Prior work tries to infer the structure of subtasks *individiually* – they do not consider the relations between compositional tasks.

We propose the *factored subtask graph inference* (FSGI) approach for tackling hierarchical and compositional tasks. We present an overview of our approach in Figure 7.1. This approach extends the problem introduced by Sohn et al. [2020]. Similar to Sohn et al. [2020], tasks are defined as factored MDPs Jonsson and Barto [2006], Boutilier et al. [1995], and we assume options Sutton et al. [1999b] (low level policies) for completing subtasks have been trained or are given as subroutines for the agent. These options are imperfect, and require certain conditions on the state to be meet before they can be successfully executed. We model the problem as a transfer RL problem. During training, an exploration policy gathers trajectories. These trajectories are then used to infer the latent *factored subtask graph*, $\hat{\mathcal{G}}$. $\hat{\mathcal{G}}$ models the hierarchies between compositional tasks and options in symbolic graph structure (shown in 7.1). In FSGI, we infer the *preconditions* of options, subtasks that must be completed before an option can be successfully executed, and the *effects* of options, subtasks that are completed after they are executed. The factored subtask graph is then used to maximize reward in the test environment by using GRProp, a method introduced by Sohn et al. [2018] which propagates a gradient through $\hat{\mathcal{G}}$ to learn the test policy.

In FSGI, we model using *factored* options and subtasks. This allows FSGI to infer the latent hierarchical and compositional structure in a *first order logic* manner. For example, in the cooking task environment in Figure 7.1 we represent all `Pickup`-object options using a *factored option*, `Pickup` x. Representing options and subtasks in factored form serves two roles: 1. The resulting graph is more **compact**. There is less redundancy when representing compositional tasks that share common structure. Hence a factored subtask graph requires less samples to infer (e.g. relations for `Pickup apple`, `Pickup pan`, etc. are inferred at once with a factored option `Pickup` x). and 2. The resulting graph can **generalize** to *unseen* subtasks, where unseen subtasks may share similar

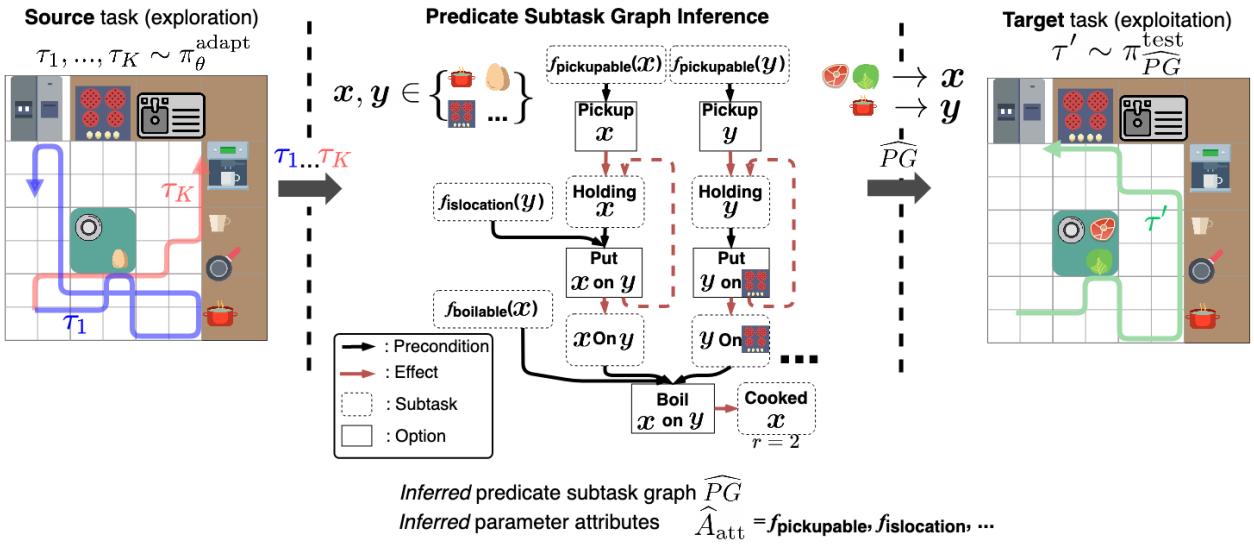


Figure 7.1: We present an overview of *factored subtask graph inference* (FSGI) in a toy cooking environment. In various tasks, the agent must cook various foods to receive reward. **Left:** The adaptation policy $\pi_\theta^{\text{adapt}}$ initially explores the cooking source task (training), generating a trajectories $\tau_1, \ldots, \tau_K$. **Middle:** Using $\tau_1, \ldots, \tau_K$, the agent infers a *factored subtask graph* $\hat{\mathcal{G}}$ of the environment, which describes the preconditions and effects between *factored* options and subtasks using (x and y) over *entities* (objects in the environment). The agent learns a set of parameter attributes ($\hat{A}_{\text{att}} = f_{\text{pickapable}} \ldots$) in a *zero-shot* manner and uses these attributes to construct $\hat{\mathcal{G}}$. **Right:** The agent initializes a separate test policy $\pi^{\text{test}}_{\hat{\mathcal{G}}}$ that maximizes reward by following the inferred factored subtask graph $\hat{\mathcal{G}}$. In this target environment (test) there exist *unseen parameters* (cabbage and meat). Preconditions and effects for these parameters are accurately inferred by substituting for entities (x and y).

structure but are not encountered during adaptation (e.g. `Pickup cabbage` in Figure 7.1).

To enable factored representation, we also learn the *attributes* of the components in the compositional tasks. These attributes are used to indicate differences in the structures of factored options and subtasks. For example, in the cooking task in Figure 7.1, not every object can be picked up with `Pickup`, so the inferred attribute $\hat{f}_{\text{pickupable}}(x)$ is a precondition to `Pickup`(x). Similarly, in a more complex cooking task, some object x may need to be sliced, before it can be boiled (e.g. cabbage), but some do not (e.g. egg). We model these structures using *parameter attributes*, $\hat{A}_{\text{att}}$ (in the cooking task case objects are parameters). We present a simple scheme to infer attributes in a *zero-shot* manner, where we infer attributes that are useful to infer relations between parameters without supervision. These attributes are then used to generalize to other parameters (or entities), that may be unseen during adaptation.

We summarize our work as follows:

- We propose the approach of *factored subtask graph inference* (FSGI) to efficiently infer the subtask structure of hierarchical and compositional tasks in a **first order logic manner**.
- We propose a simple *zero-shot* learning scheme to infer *entity attributes*, which are used to relate the structures of compositional subtasks.
- We demonstrate FSGI on a symbolic cooking environment that has complex hierarchical and compositional task structure. We show FSGI can accurately infer this structure more *efficiently* than prior work and *generalize* this structure to unseen tasks.

7.2 Problem Definition

7.2.1 Background: Transfer Reinforcement Learning

A *task* is characterized by an MDP $\mathcal{M}_G = \langle \mathcal{A}, \mathcal{S}, \mathcal{T}_G, \mathcal{R}_G \rangle$, which is parameterized by a task-specific G, with an action space $\mathcal{A}$, state space $\mathcal{S}$, transition dynamics $\mathcal{T}_G$, and reward function $\mathcal{R}_G$. In the transfer RL formulation Duan et al. [2016], Finn et al. [2017], an agent is given a fixed distribution of training tasks $\mathcal{M}^{\text{train}}$, and must learn to efficiently solve a distribution of unseen test tasks $\mathcal{M}^{\text{test}}$. Although these distributions are disjoint, we assume there is some similarity between tasks such that some learned behavior in training tasks may be useful for learning test tasks. In each task, the agent is given k timesteps to interact with the environment (the *adaptation* phase), in order to adapt to the given task. After, the agent is evaluated on its adaptation (the *test* phase). The agent's performance is measured in terms of the expected return:

$$\mathcal{R}_{\mathcal{M}_G} = \mathbb{E}_{\pi_k, \mathcal{M}_G} \left[\sum_{t=1}^{H} r_t \right] \tag{7.1}$$

where π_K is the policy after k timesteps of the adaptation phase, H is the horizon in the test phase, and r_t is the reward at time t of the test phase.

7.2.2 Background: The Subtask Graph Problem

The subtask graph inference problem is a transfer RL problem where tasks are parameterized by hierarchies of *subtasks* Sohn et al. [2020], Φ. A task is composed of N subtasks, $\{\Phi^1, \dots, \Phi^N\} \subset \Phi$, where each subtask $\Phi \in \Phi$ is parameterized by the tuple $\langle \mathcal{S}_{\text{comp}}, G_r \rangle$, a *completion set* $\mathcal{S}_{\text{comp}} \subset \mathcal{S}$, and a *subtask reward* $G_r : \mathcal{S} \to \mathbb{R}$. The completion set $\mathcal{S}_{\text{comp}}$ denotes whether the subtask Φ is *complete*, and the subtask reward G_r is the reward given to the agent when it completes the subtask.

Following Sohn et al. [2020], we assume the agent learns a set of options $\mathbb{O} = \{\mathcal{O}^1, \mathcal{O}^2, \dots\}$ that *execute* different subtasks Sutton et al. [1999b]. These options can be learned by conditioning on subtask goal reaching reward: $r_t = \mathbb{I}(s_t \in \mathcal{S}^i_{\text{comp}})$. Each option $\mathcal{O} \in \mathbb{O}$ is parameterized by the tuple $\langle \pi, G_{\text{prec}}, G_{\text{effect}} \rangle$. There is a trained policy π corresponding to each $\mathcal{O}$. These options may be *eligible* at different precondition states $G_{\text{prec}} \subset \mathcal{S}$, where the agent must be in certain states when executing the option, or the policy π fails to execute (also the *initial set* of $\mathcal{O}$ following Sutton et al. [1999b]). However, unlike Sohn et al. [2020], these options may complete an unknown number of subtasks (and even *un*complete subtasks). This is parameterized by $G_{\text{effect}} \subset \mathcal{S}$ (also the *termination set* of $\mathcal{O}$ following Sutton et al. [1999b]).

Environment: We assume that the subtask completion and option eligibility is known to the agent. (But the precondition, effect, and reward is hidden and must be inferred). In each timestep t the agent is the state $s_t = \{x_t, e_t, \text{step}_t, \text{step}_{\text{phase},t}, \text{obs}_t\}$.

- **Completion:** $x_t \in \{0, 1\}^N$ denotes which subtasks are complete.
- **Eligibility:** $e_t \in \{0, 1\}^M$ denotes which options are eligible.
- **Time Budget:** $\text{step}_t \in \mathbb{Z}_{>0}$ is the number steps remaining in the episode.
- **Adaptation Budget:** $\text{step}_{\text{phase},t} \in \mathbb{Z}_{>0}$ is the number steps remaining in the adaptation phase.
- **Observation:** $\text{obs}_t \in \mathbb{R}^d$ is a low level observation of the environment at time t.

7.2.3 The Factored Subtask Graph Problem

Subtasks and Option Entities In the real world, compositional subtasks can be described in terms of a set of *entities*, $\mathcal{E}$. (e.g. `pickup`, `apple`, `pear`, $\dots \in \mathcal{E}$) that can be recombined to form new subtasks (e.g. (`pickup`, `apple`), and (`pickup`, `pear`)). We assume that these entities are given

to the agent. Similarly, the learned options that execute these subtasks can also be parameterized by the same entities (e.g. [`pickup`, `apple`], and [`pickup`, `pear`]).

In real world tasks, we expect learned options with entities that share "attributes" to have similar policy, precondition, and effect, as they are used to execute subtasks with similar entities. For example, options [`pickup`, `apple`], and [`pickup`, `pear`] may share the same policy, where the policy's motor control is the same but has a different target entity (*apple* or *pear*). *apple* and *pear* share the same attribute *pickupable*; they are both entities that can be picked up. In another example, options [`cook`, `egg`, `pot`] and [`cook`, `cabbage`, `pot`] share similar preconditions (the target ingredient must be placed in the *pot*), but also different (*cabbage* must be sliced, but the egg does not). In this example, *egg* and *cabbage* are both *boilable* entities, but *egg* is not *sliceable*.

To model these similarities, we assume in each task, there exist boolean *latent attribute functions* which indicate shared attributes in entities. E.g. $f_{\text{pickapable}} : \mathcal{E} \to \{0, 1\}$, where $f_{\text{pickapable}}(\texttt{apple}) = 1$. We will later try to infer the values of these latent entities, so we additionally assume there exist some weak supervision, where a low-level embedding of entities is provided to the agent, $f_{\text{entityembed}} : \mathcal{E} \to \mathbb{R}^D$.

The Factored Subtask Graph Our goal is to infer the underlying task structure between subtasks and options so that the agent may complete subtasks in an optimal order. As defined in the previous sections, this task structure can be completely determined by the option preconditions, option effects, and subtask rewards. As such we define the *factored subtask graph* to be the tuple of the *factored* preconditions, effects, and rewards for all subtasks and options:

$$\mathcal{G} = \langle \mathcal{G}_{\text{pcond}}, \mathcal{G}_{\text{eff}}, \mathcal{G}_r \rangle \tag{7.2}$$

where

$$\mathcal{G}_{\text{pcond}} : \mathcal{E}^N \times \mathcal{P}(\Phi) \to \{0, 1\} \tag{7.3}$$

$$\mathcal{G}_{\text{eff}} : \mathcal{E}^N \times \mathcal{P}(\Phi) \to \mathcal{P}(\Phi) \tag{7.4}$$

$$\mathcal{G}_r : \mathcal{E}^N \times \mathcal{P}(\Phi) \to \mathbb{R} \tag{7.5}$$

The factored precondition, $\mathcal{G}_{\text{pcond}}$, is a function from an option with N entities and a subtask completion set to $\{0, 1\}$, which specifies whether the option is eligible under a completion set. E.g. If $\mathcal{G}_{\text{pcond}}([X_1, X_2], \{\Phi^1, \Phi^2\}) = 1$, then option $[X_1, X_2]$ is eligible if Φ^1 and Φ^2 are complete. The factored effect, $\mathcal{G}_{\text{eff}}$, is a function from an option with N entities and subtask completion set to a different completion set. E.g. If $\mathcal{G}_{\text{eff}}([X_1, X_2], \{\Phi^1, \Phi^2\}) = \{\Phi^1, \Phi^3\}$, then executing an option with parameters $[X_1, X_2]$ on a state with subtask completion $\{\Phi^1, \Phi^2\}$ completes Φ^3 and *un*completes Φ^2. Finally, the factored reward, $\mathcal{G}_r$, is a function from a subtask with N entities to the reward given to the agent from executing that subtask.

Our previous assumption that options with similar entities and attributes share preconditions

and effects manifests in $\mathcal{G}_{\text{pcond}}$ and $\mathcal{G}_{\text{eff}}$ where these functions tend to be *smooth*. Similar inputs to the function (similar option entities) tend to yield similar output (similar eligibility and effect values). This smoothness gives two benefits. 1. We can share experience between similar options for inferring preconditions and effect. 2. This enables generalization to preconditions and effects of unseen entities. Note that this smoothness does *not* apply to the reward $\mathcal{G}_r$. We assume reward given for subtask completion is independent across tasks.

In our experiments, we assume the first entity of every subtask and option serves as a "verb" entity (e.g. `pickup`, `cook`, etc.). We assume there is there is no shared structure across subtasks and options with different verbs.

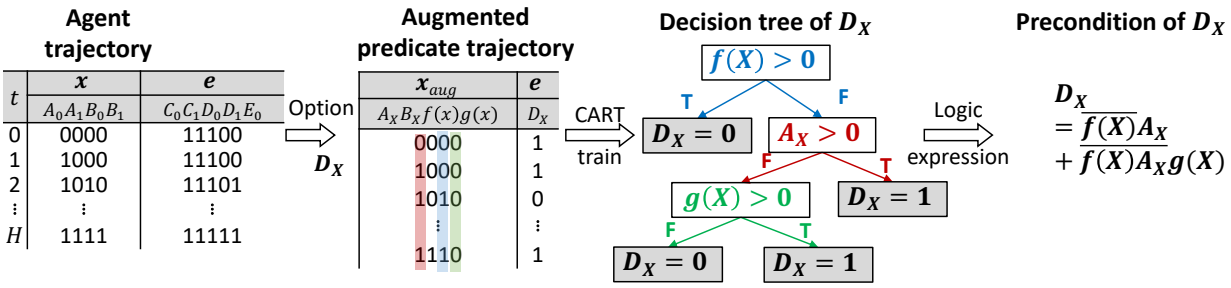
7.3 Method

We propose the *Factored Subtask Graph Inference* (FSGI) method to efficiently infer the latent factored subtask graph $\mathcal{G} = \langle \mathcal{G}_{\text{pcond}}, \mathcal{G}_{\text{eff}}, \mathcal{G}_r \rangle$. Figure 7.2 gives an overview of our approach. At a high level, we use the adaptation phase to gather *adaptation trajectories* from the environment using an adaptation policy $\pi_\theta^{\text{adapt}}$. Then, we use the adaptation trajectories to infer the latent subtask graph $\hat{\mathcal{G}}$. In the test phase, a *test policy* $\pi_{\hat{\mathcal{G}}}^{\text{test}}$ is conditioned on the inferred subtask graph $\hat{\mathcal{G}}$ and maximizes the reward. As the performance of the test policy is dependent on the inferred subtask graph $\hat{\mathcal{G}}$, it is important to accurately infer this graph. Note that the test task may contain subtasks that are *unseen* in the training task. We learn a predicate subtask graph $\hat{\mathcal{G}}$ that can *generalize* to these unseen subtasks and options.

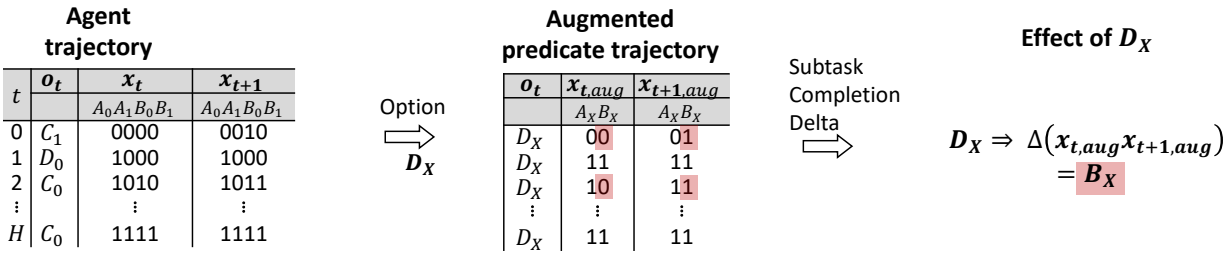
7.3.1 Zero-shot Learning Entity Attributes

Recall from Section 7.2.3 that we assume there exist *latent attributes* that indicate shared structure between options and subtasks with the same attributes. E.g. One attribute may be $f_{\text{pickapable}} : \mathcal{E} \to \{0, 1\}$, where $f_{\text{pickapable}}(\texttt{apple}) = 1$, etc. Our goal is to infer a set of candidate attribute functions, $\hat{A}_{\text{att}} = \{\hat{f}_1, \hat{f}_2, \dots\}$, such that options with the same attributes indicates the same preconditions. As there is no supervision involved, we formulate this inference as a *zero shot learning problem* Palatucci et al. [2009]. Note the inferred attributes that are preconditions for options should not only construct an accurate predicate subtask graph for options seen in the adaptation phase, but also unseen options.

During the adaptation phase, the agent will encounter a set of seen entities $E \subset \mathcal{E}$. We construct candidate attributes from E using our *smoothness* assumption, where similar entities result in similar preconditions. We generate candidate attributes based on similarity using the given entity embedding, $f_{\text{entityembed}} : \mathcal{E} \to \mathbb{R}^D$.



(a) Factored Precondition Inference via inductive logic programming. We run precondition inference for every option and show D_X as an example. **1.** The first table is built from the agent's trajectory (x is the subtask completion, e the option eligibility). **2.** We build the second table, the "augmented" trajectory by substituting X into all possible subtask completions, A_X, B_X, and inferred attributes f, g. **3.** We train a decision tree over the table, to infer the relation $x_{\text{aug}} \to D_x$. **4.** We translate the decision tree into an equivalent predicate boolean expression, which is one part of the inferred factored subtask graph $\hat{\mathcal{G}}$.



(b) Factored Effect Inference. We run effect inference for every option and D_X as an example. **1.** The first table is built from the agent's trajectory (x_t, o_t is the subtask completion and option executed at time t). **2.** We build the second table, the "augmented" trajectory by substituting X into all possible subtask completions, A_X, B_X, and restricting the table to only row where $o_t = D_X$. **3.** We infer option dynamics, $(x_t, o_t) \mapsto x_{t+1}$, by calculating the simple aggregated difference between subtask completion before and after D_X, $\Delta(x_{t,\text{aug}}, x_{t+1,\text{aug}})$.

Figure 7.2: An overview of our approach for estimating the factored subtask graph $\hat{\mathcal{G}}$ in a simple environment with subtasks A, B and options C, D, E. Each subtask and option has a parameter 0 or 1. Note by inferring the factored precondition and effects, we can infer the behavior unseen subtasks and options such as D_2.

Let $C = \{C_1, C_2, \dots\}$ be an exhaustive set of clusters generated from E using $f_{\text{entityembed}}$. Then, we define a candidate attribute function from each cluster.

$$\hat{f}_i(X) := \mathbb{I}[X \in C_i] \tag{7.6}$$

To infer the attribute of an unseen entity $X \notin E$, we use a 1-Nearest Neighbor classifier that uses the attributes of the nearest seen entity Fix [1985].

$$\hat{f}_i(X) = \mathbb{I}[X^* \in C_i] \tag{7.7}$$

where $X^* = \arg\min_{X' \in E} \text{distance}(f_{\text{entityembed}}(X), f_{\text{entityembed}}(X'))$.

7.3.2 Factored Subtask Graph Inference

Let $\tau_H = \{s_1, o_1, r_1, d_1, \dots, s_H\}$ be the adaptation trajectory of the adaptation policy $\pi_\theta^{\text{adapt}}$ after H time steps. Our goal is to infer the maximum likelihood factored subtask graph $\mathcal{G}$ given this trajectory τ_H.

$$\hat{\mathcal{G}}^{\text{MLE}} = \underset{\mathcal{G}_{\text{pcond}}, \mathcal{G}_{\text{eff}}, \mathcal{G}_r}{\arg\max}\ p(\tau_H | \mathcal{G}_{\text{pcond}}, \mathcal{G}_{\text{eff}}, \mathcal{G}_r) \tag{7.8}$$

We can expand the likelihood term as:

$$p(\tau_H | \mathcal{G}_{\text{pcond}}, \mathcal{G}_{\text{eff}}, \mathcal{G}_r) \tag{7.9}$$

$$= p(s | \mathcal{G}_{\text{pcond}}, \mathcal{G}_{\text{eff}}) \prod_{t=1}^{H} \pi_\theta(o_t | \tau_t) p(s_{t+1} | s_t, o_t, \mathcal{G}_{\text{pcond}}, \mathcal{G}_{\text{eff}}) p(r_t | s_t, o_t, \mathcal{G}_r) p(d_t | s_t, o_t) \tag{7.10}$$

$$\propto p(s | \mathcal{G}_{\text{pcond}}, \mathcal{G}_{\text{eff}}) \prod_{t=1}^{H} p(s_{t+1} | s_t, o_t, \mathcal{G}_{\text{pcond}}, \mathcal{G}_{\text{eff}}) p(r_t | s_t, o_t, \mathcal{G}_{\text{eff}}, \mathcal{G}_r) \tag{7.11}$$

where we dropped terms independent of $\mathcal{G}$. From section 7.2.2 and 7.2.3, the predicate precondition $\mathcal{G}_{\text{pcond}}$ determines the mapping from completion x to option eligibility e, $x \mapsto e$, the predicate effect $\mathcal{G}_{\text{eff}}$ determines the mapping from completion and option to completion, $(x_t, o) \mapsto x_{t+1}$, and finally, the predicate reward $\mathcal{G}_r$ determines the reward given when a subtask is completed at time t. Then, we can define the MLE as:

$$\hat{PG}^{\text{MLE}} = \left(\hat{\mathcal{G}}^{\text{MLE}}_{\text{pcond}}, \hat{\mathcal{G}}^{\text{MLE}}_{\text{eff}}, \hat{\mathcal{G}}^{\text{MLE}}_{r} \right) \tag{7.12}$$

$$= \Bigg(\underset{\mathcal{G}_{\text{pcond}}}{\arg\max} \prod_{t=1}^{H} p(e_t | x_t, \mathcal{G}_{\text{pcond}}), \underset{\mathcal{G}_{\text{eff}}}{\arg\max} \prod_{t=1}^{H} p(x_{t+1} | x_t, o_t, \mathcal{G}_{\text{eff}}), \tag{7.13}$$

$$\underset{\mathcal{G}_r}{\arg\max} \prod_{t=1}^{H} p(r_t | o_t, o_{t+1}, \mathcal{G}_r) \Bigg) \tag{7.14}$$

Next, we explain how to compute $\hat{\mathcal{G}}_{\text{pcond}}$, $\hat{\mathcal{G}}_{\text{eff}}$, and $\hat{\mathcal{G}}_r$.

Factored Precondition Inference via Predicate Logic Induction We give an overview of how we infer the option preconditions $\hat{\mathcal{G}}_{\text{pcond}}$ in Figure 7.2a. Note from the definition from 7.2.3, we can view the precondition $\mathcal{G}_{\text{pcond}}$ as a deterministic function, $f_{\mathcal{G}_{\text{pcond}}} : (E, x) \mapsto \{0, 1\}$, where E is the option entities, and x is the completion set vector. Hence, the probability term in Eq.(7.13) can be written as $p(e_t|x_t, \mathcal{G}_{\text{pcond}}) = \prod_{i=1}^{N} \mathbb{I}[e_t^{(i)} = f_{\mathcal{G}_{\text{pcond}}}(E^{(i)}, x_t)]$ where $\mathbb{I}$ is the indicator function, and $E^{(i)}$ is the entity set of the ith option in the given task. Thus, we have

$$\hat{\mathcal{G}}_{\text{pcond}}^{\text{MLE}} = \arg\max_{\mathcal{G}_{\text{pcond}}} \prod_{t=1}^{H} \prod_{i=1}^{N} \mathbb{I}[e_t^{(i)} = f_{\mathcal{G}_{\text{pcond}}}(E^{(i)}, x_t)] \tag{7.15}$$

Following Sohn et al. [2020], this quantity can maximized by finding a boolean function $\hat{f}_{\mathcal{G}_{\text{pcond}}}$ over *only* subtask completions x_t that satisfies all the indicator functions in Eq.(7.15). However this yields multiple possible solutions — particularly the preconditions of unseen option entities in the trajectory τ_H. If we infer a boolean function separately over all seen options (without considering the option parameters), this solution is identical to the solution proposed by Sohn et al. [2020]. We want to additionally generalize our solution over multiple unseen subtasks and options using the entities, E.

We leverage our smoothness assumption — that $\hat{f}_{\mathcal{G}_{\text{pcond}}}$ is *smooth* with respect to the input entities and attributes. E.g. If the inferred precondition for the option [`pickup`, X] is the candidate attribute $\hat{f}(X)$, any entity X where $\hat{f}(X) = 1$ has the same precondition. I.e. For some unseen entity set E^* we want the following property to hold:

$$\hat{f}_i(E) = \hat{f}_i(E^*) \text{ for some } i \;\Rightarrow \hat{f_{\mathcal{G}_{\text{pcond}}}}(E, x_t) = \hat{f_{\mathcal{G}_{\text{pcond}}}}(E^*, x_t) \tag{7.16}$$

To do this, we infer a boolean function $\hat{f_{\mathcal{G}_{\text{pcond}}}}$ over *both* subtask completions x_t *and* entity variables $X \in E$. We use (previously inferred) candidate attributes over entities, $\hat{f}_i(X) \forall X \in E$ in the boolean function to serve as *quantifiers*. Inferring in this manner insures that the precondition function $\hat{f}_{\mathcal{G}_{\text{pcond}}}$ is *smooth* with respect to the input entities and attributes. Note that some but not all attributes may be shared in entities. E.g. [`cook`, `cabbage`] has similar but not the same preconditions as [`cook`, `egg`]. So, we cannot directly reuse the same preconditions for similar entities. We want to generalize between different *combinations* of attributes.

We translate this problem as an *inductive logic programming* (ILP) problem Muggleton and De Raedt [1994]. We infer the eligibility (boolean output) of some option $\mathcal{O}$ with some entities(s) $E = \{X_1, X_2, \dots\}$, from boolean input formed by all possible completion values $\{x_t^i\}_{t=1}^{H}$, and all attribute values $\{\hat{f}_i(X)\}_{X \in E}^{i=1,..}$. We use the *classification and regression tree* (CART) with Gini impurity to infer the the precondition functions $\hat{f}_{\mathcal{G}_{\text{pcond}}}$ for each parameter E Breiman [1984]. Finally, the inferred decision tree is converted into an equivalent symbolic logic expression and used to build the factored subtask graph.

Factored Effect Inference We give an overview of how we infer the option effects $\hat{\mathcal{G}}_{\text{eff}}$ in Figure 7.2b. From the definitions from section 7.2.3, we can write the predicate option effect $\mathcal{G}_{\text{eff}}$ as a deterministic function $f_{\mathcal{G}_{\text{eff}}} : (E, x_t) \mapsto x_{t+1}$, where if there is subtask completion x_t, executing option $\mathcal{O}$ (with entities E) successfully results in subtask completion x_{t+1}. Similar to precondition inference, we have

$$\hat{\mathcal{G}}_{\text{eff}}^{\text{MLE}} = \arg\max_{\mathcal{G}_{\text{eff}}} \prod_{t=1}^{H} \prod_{i=1}^{N} \mathbb{I}[x_{t+1} = f_{\mathcal{G}_{\text{eff}}}(E^{(i)}, x_t)] \tag{7.17}$$

As this is deterministic, we can calculate the element-wise difference between x_t (before option) and x_{t+1} (after option) to infer $f_{\mathcal{G}_{\text{eff}}}$.

$$\hat{f_{\mathcal{G}_{\text{eff}}}}(E^{(i)}, x) = x + \mathbb{E}_{t=1\ldots H}[x_{t+1} - x_t | o_t = \mathcal{O}^{(i)}] \tag{7.18}$$

Similar to precondition inference, we also want to infer the effect of options with unseen parameters. We leverage the same smoothness assumption:

$$\hat{f}_i(E) = \hat{f}_i(E^*) \text{ for some } i \;\Rightarrow \hat{f_{\mathcal{G}_{\text{eff}}}}(E, x_t) = \hat{f_{\mathcal{G}_{\text{eff}}}}(E^*, x_t) \tag{7.19}$$

Unlike preconditions, we expect the effect function to be relatively constant across attributes, i.e., the effect of executing option [`cook`, X] is always completing the subtask (`cooked`, X), no matter the attributes of X. So we directly set the effect of unseen entities, $\hat{f_{\mathcal{G}_{\text{eff}}}}(E^*, x_t)$, by similarity according to Equation 7.19.

Reward Inference We model the subtask reward as a Gaussian distribution $\mathcal{G}_r(E) \sim \mathcal{N}(\hat{\mu}_E, \hat{\sigma}_E$. The MLE estimate of the subtask reward becomes the empirical mean of the rewards received during the adaptation phase when subtask with parameter $\mathcal{T}$ becomes complete. For the ith subtask in the task with entities E^i,

$$\hat{\mathcal{G}}_r(E^i) = \hat{\mu}_{E^i} = \mathbb{E}_{t=1\ldots N}[r_t | x_{t+1}^i - x_t^i = 1] \tag{7.20}$$

Note we do not use the smoothness assumption for $\hat{\mathcal{G}}_r(E)$, as we assume reward is independently distributed across tasks. We automatically set $\hat{\mathcal{G}}_r(E^*) = 0$ for unseen subtasks with entities E^*.

7.3.3 Task Transfer and Adaptation

In the test phase, we instantiate a *test policy* $\pi^{\text{test}}_{\hat{\mathcal{G}}_{\text{prior}}}$ using the inferred factored subtask graph $\hat{\mathcal{G}}_{\text{prior}}$, inferred from samples gathered from the training task. The goal of the test policy is to maximize reward in the test environment using $\hat{\mathcal{G}}_{\text{prior}}$. As we assume the reward is independent across tasks, we re-estimate the reward of the test task according to Equation 7.20, without task transfer. With the reward inferred, this yields the same problem setting given in Sohn et al. [2018]. Sohn et al. [2018] tackle this problem using GRProp, which models the subtask graph as differentiable function over reward, so that the test policy has a dense signal on which options to execute are likely to maximally

increase the reward.

However, the inferred factored subtask graph may be imperfect, the inferred precondition and effects may not transfer to the test task. To adapt to possibly new preconditions and effects, we use samples gathered in the adaptation phase of the test task to infer a new factored subtask graph $\hat{\mathcal{G}}_{\text{test}}$, which we use to similarly instantiate another test policy $\pi^{\text{test}}_{\hat{\mathcal{G}}_{\text{test}}}$ using GRProp. We expect $\hat{\mathcal{G}}_{\text{test}}$ to eventually be more accurate than $\hat{\mathcal{G}}_{\text{prior}}$ as more timesteps are gathered in the test environment. Though, early on, $\hat{\mathcal{G}}_{\text{prior}}$ may be more accurate.

To maximize performance on test, we thus choose to instantiate a posterior test policy $\pi^{\text{test}}_{\text{posterior}}$, which is an *ensemble* policy over $\pi^{\text{test}}_{\hat{\mathcal{G}}_{\text{prior}}}$ and $\pi^{\text{test}}_{\hat{\mathcal{G}}_{\text{test}}}$. We heuristically set the weights of $\pi^{\text{test}}_{\text{posterior}}$ to favor $\pi^{\text{test}}_{\hat{\mathcal{G}}_{\text{prior}}}$ early in the test phase, and $\pi^{\text{test}}_{\hat{\mathcal{G}}_{\text{test}}}$ later in the test phase.

7.4 Related Work

Subtask Graph Inference. The subtask graph inference (SGI) framework Sohn et al. [2018, 2020] assumes that a task consists of multiple base subtasks, such that the entire task can be solved by completing a set of subtasks in the right order. Then, it has been shown that SGI can efficiently solve the complex task by explicitly inferring the precondition relationship between subtasks in the form of a graph using an inductive logic programming (ILP) method. The inferred subtask graph is in turn fed to an execution policy that can predict the optimal sequence of subtasks to be completed to solve the given task.

However, the proposed SGI framework is limited to a single task; the knowledge learned in one task cannot be transferred to another. This limits the SGI framework such that does not scale well to compositional tasks, and cannot generalize to unseen tasks. We extend the SGI framework by modeling *factored* subtasks and options, which encode relations between tasks to allow efficient and general learning. In addition, we generalize the SGI framework by learning an effect model – In the SGI framework it was assumed that for each subtask there is a corresponding option, that completes that subtask (and does not effect any other subtask). This assumption is unrealistic when there options must also be parameterized in the combinatorial task structure, and there exist subtasks that cannot be completed *without un-completing another* (e.g. `Pickup` and `Place` in Figure 7.1).

Compositional Task Generalization. Prior work has also tackled compositional generalization in a symbolic manner Loula et al. [2018], Andreas et al. [2017], Oh et al. [2017]. Loula et al. [2018] test compositional generalization of natural language sentences in recurrent neural networks. Andreas et al. [2017], Oh et al. [2017] tackle compositional task generalization in an *instruction following* context, where an agent is given a natural language instruction describing the task the agent must complete (e.g. "pickup apple"). These works use *analogy making* to learn policies that

can execute instructions by analogy (e.g. "pickup X"). However, these works construct policies on the *option level* – they construct policies that can execute "pickup X" on different X values. They also do not consider hierarchical structure for the order which options should be executed (as they are given the option order through instruction). Our work aims to learn these analogy-like relations at a between-options level, where certain subtasks and options must be completed before another option can be executed.

Classical Planning. At a high level, a factored subtask graph $\mathcal{G}$ is equivalent to a STRIPS planning domain Fikes and Nilsson [1971] with an attribute model add-on Frank and Jónsson [2003]. *Operators* in STRIPS correspond to options in $\mathcal{G}$. The symbolic state space in STRIPS corresponds to subtask completion in $\mathcal{G}$. Arguments to operators and states are similar to parameters in $\mathcal{G}$. Most work in classical planning focuses on how to use a domain specification to instantiate a solution plan, equivalent to our reward maximization phase. A viable option FSGI is to replace our test policy with a classical planner using our inferred factored subtask graph.

Prior work in classical planning has proposed to learn STRIPS domain specifications (action schemas) through given trajectories (action traces) Suárez-Hernández et al. [2020], Mehta et al. [2011], Walsh and Littman [2008], Zhuo et al. [2010]. Our work differs from these in 3 major ways: 1. FSGI learns an *attribute* model, which is crucial to generalizing compositional tasks with components of different behaviors. 2. We evaluate FSGI on more hierarchical domains, where prior work has evaluated on pickup-place/travelling classical planning problems, which admit flat structure. 3. We evaluate FSGI on generalization, where there may exist subtasks and options that are not seen during adaptation.

7.5 Experiments

We aim to answer the following questions:

1. Can PSGI *generalize* to unseen evaluation tasks in zero-shot manner by transferring the inferred the latent task structure?

2. Does PSGI *efficiently* infers the latent task structure compared to prior work (MSGI Sohn et al. [2020])?

3. Can PSGI *generalize* to unseen subtasks with unseen entities using the inferred attributes?

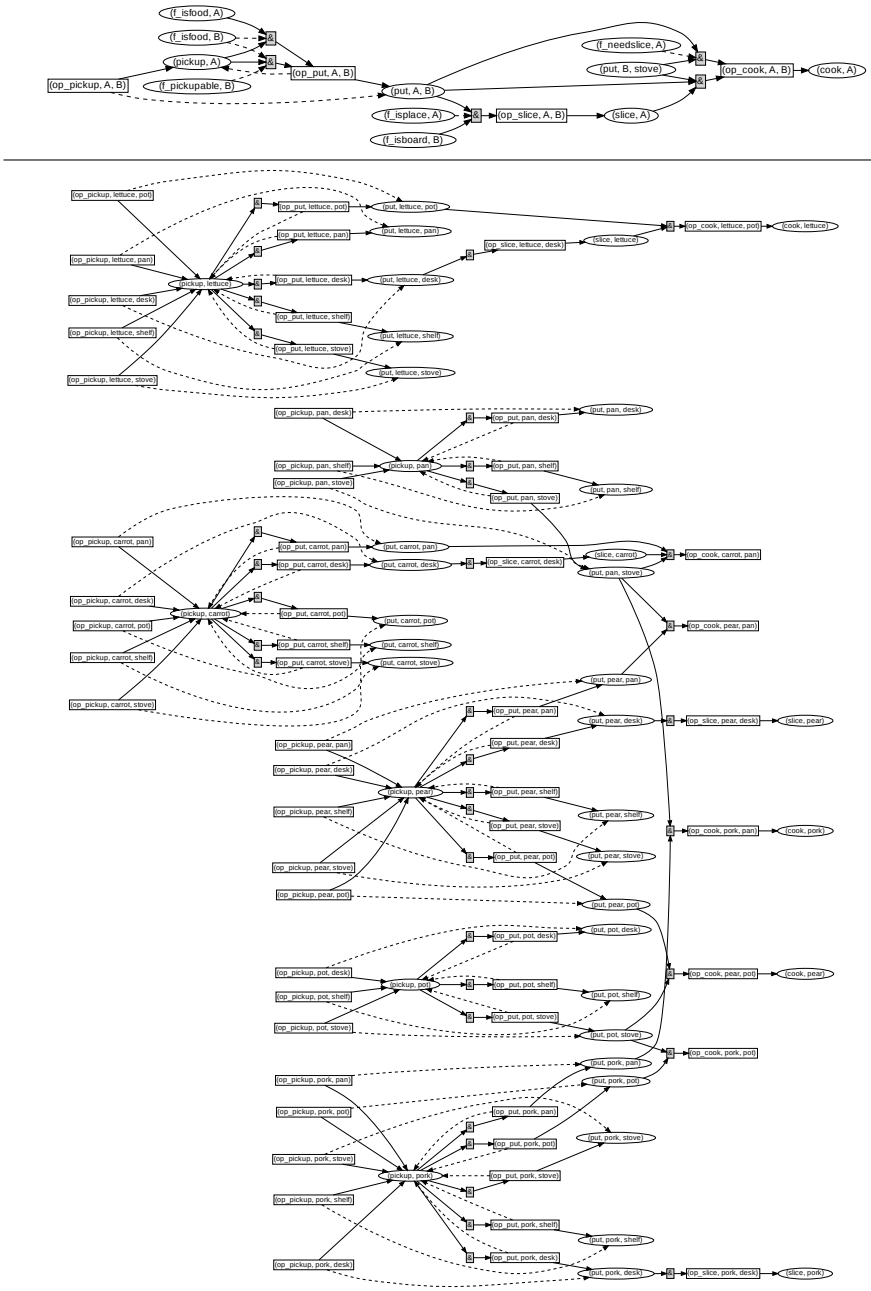


Figure 7.3: (Top) the inferred factored subtask graph $\hat{PG}$ by PSGI and (bottom) the inferred subtask graph $\hat{G}$ by **MSGI**$^+$ after 2000 timesteps in the **Cooking** environment. For **MSGI**$^+$, 262 options with no inferred precondition and effect were not visualized for readability. Options are represented in rectangular nodes. Subtask completions and attributes are are in oval nodes. A solid line represents a positive precondition / effect, dashed for negative.

7.5.1 Environment

We evaluate PSGI in a novel symbolic environment, **Cooking** and **Mining**. An example of the simplified **Cooking** task is shown in Figure 7.1. For each environment, we manually define the factored subtask graph in terms factored subtasks, options, and attributes in first-order logic form. Then, the subtask graph can be built by replacing each predicate with a set of task-specific entities. Each task is randomized by randomly sampling the entities from the entity pool. The entity pool for training and evaluation tasks are different such that the agent should be able to generalize to unseen entities to solve the evaluation tasks in zero-shot setting.

Tasks. Cooking environment has a pool of 22 entities and 10 entities are chosen at random for each task. Then, the subtasks and options are populated by replacing the parameters in factored subtasks and options by the sampled entities; *e.g.*, we replace `X` and `Y` in the factored subtask `(pickup, X, Y)` by {`apple, cabbage, table`} to populate nine subtasks. This results in around 320 options and 100 subtasks. The ground-truth attributes are also predefined in the templates, but is not available to the agent. Similarly for **Mining**, we randomly sample 12 entities from a pool of 18 entities and populate around 180 subtasks and 180 options for each task. The reward is assigned at random to one of the subtasks that have the largest critical path length, where the critical path length is the minimum number of options to be executed to complete each subtask.

Observations. At each time step, the agent observes the completion and eligibility vectors (see Section 7.2.2 for definitions) and the embeddings of corresponding subtasks and options. The subtask and option embeddings are the concatenated vector of the embeddings of its entities; *e.g.*, for `pickup, apple, table` the embedding is $[f(\texttt{pickup}), f(\texttt{apple}), f(\texttt{table})]$ where $f(\cdot)$ can be an image or language embeddings. For **Mining** and **Cooking** environment, we choose to use 50 dimensional GloVE word embeddings from Pennington et al. [2014] as the embedding function $f(\cdot)$.

7.5.2 Baselines

- **MSGI**$^+$ is the **MSGI** Sohn et al. [2020] agent modified to be capable of solving our **Cooking** and **Mining** tasks. We augmented **MSGI** with an effect model, separate subtasks and options in the ILP algorithm, and replace the GProp with our cyclic GRProp.
- **RL**^{2} Duan et al. [2016] trains a recurrent model over the training tasks to quickly adapt to given task as it is rolled out.
- **HRL** Andreas et al. [2017][1] is the option-based hierarchical reinforcement learning agent. It is an actor-critic model over the pre-learned options.

[1] In Andreas et al. [2017] this agent was referred as Independent model.

- **Random** agent uniformly randomly executes any eligible option.

We meta-train **RL**^{2} and **FSGI** on training tasks and meta-eval on evaluation tasks to test its adaptation efficiency and generalization ability. We train **HRL** on evaluation tasks to test its adaptation (*i.e.*, learning) efficiency. We evaluate **Random** baseline on evaluation tasks to get a reference performance. We use the same recurrent neural network with self-attention-mechanism so that the agent can handle varying number of (unseen) parameterized subtasks and options depending on the tasks.

7.5.3 Zero-/Few-shot Transfer Learning Performance

Zero-shot learning performance. Figure 7.4 shows the zero-shot and few-shot transfer learning performance of the compared methods on **Mining** and **Cooking** domains. First, **FSGI** achieves over 50 and 30% success rate on **Cooking** and **Mining** domain without observing any samples (*i.e.*, x-axis value = 0) in unseen evaluation tasks. This indicates that the factored subtask graph effectively captures the shared task structure, and the inferred attributes generalizes well to unseen entities in zero-shot manner. Note that **MSGI**$^+$, **HRL**, and **Random** baselines have no ability to transfer its policy from training tasks to unseen evaluation tasks. We note that **MSGI**$^+$ has no mechanism for generalizing to the tasks with unseen entities.

Few-shot learning performance. In Figure 7.4, **FSGI** achieves over 90 and 80% success rate on **Cooking** and **Mining** domains respectively after only 1000 steps of adaptation, while other baselines do not learn any meaningful policy except **MSGI**$^+$ in cooking environment. This demonstrates that the factored subtask graph enables **FSGI** to share the experience of similar subtasks and options (*e.g.*, `pickup X on Y` for all possible pairs of `X` and `Y`) such that the sample efficiency is increased by roughly the factor of number of entities compared to using subtask graph in **MSGI**$^+$.

7.5.4 Comparison on Task Structure Inference

We ran **FSGI** and **MSGI**$^+$ in the **Cooking** and **Mining** environment, inferring the latent subtask graphs every 50 timesteps, for 2000 timesteps. The inferred graphs at 2000 timesteps are shown in Figure 7.3. **FSGI** infers the factored graph using first-order logic, and thus it is more compact (XX nodes and XX edges). On the other hand, **MSGI**$^+$ infers the subtask graph without factoring out the shared structure, resulting in non-compact graph with hundreds of subtasks and options. Moreover, graph inferred by **FSGI** has 0% error in precondition and effect model inference. The graph inferred by **MSGI**$^+$ has 38% error in the preconditions (the six options that **MSGI**$^+$ completely failed to infer any precondition are not shown in the figure for readability). The result shows that the

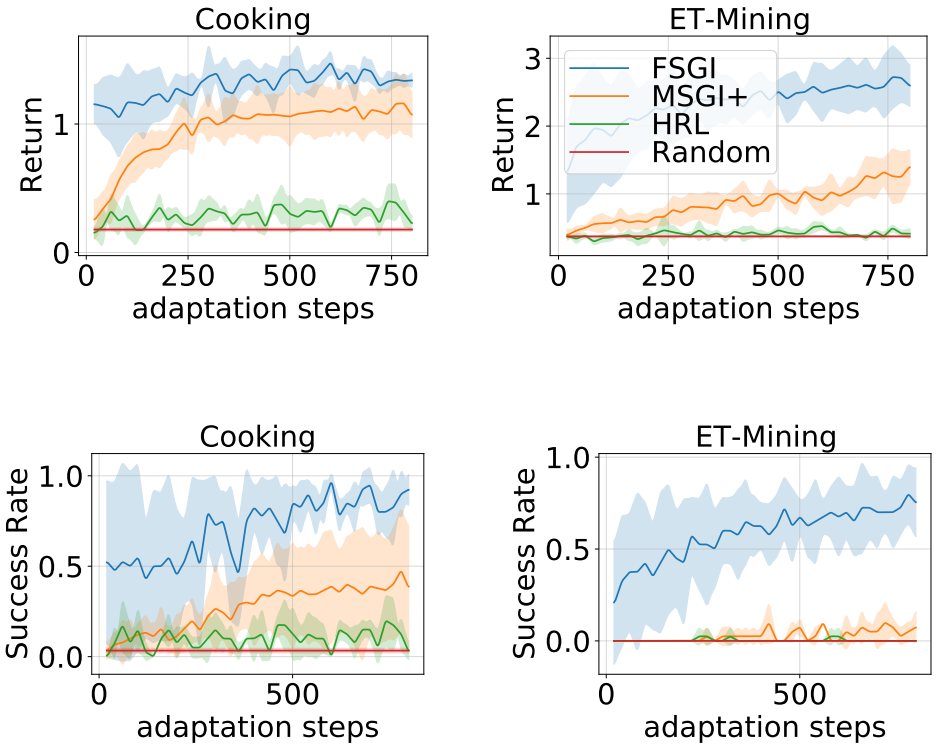


Figure 7.4: The adaptation curve on **Cooking** and **Mining** domains in terms of success rate and return.

factored graph can infer the task structure much more efficiently than the subtask graph by sharing the experiences of different options using the factored options and subtasks.

7.5.5 Generalization of Attributes to the Unseen Entities

As described in section 7.3.1, (when attributes are not provided), we infer attributes from an exhaustive powerset of all possible features on seen parameters. The attributes that are used for the graph are then likely to be semantically meaningful, as the decision tree selects the most efficient features. Hence, to test whether PSGI is generalizable, we can evaluate *whether attributes are accurately inferred* for unseen parameters when only given the ground truth attributes on seen parameters (given that PSGI will infer the ground truth for the seen parameters).

We measure the generalization error of PSGI if some "weak" signal is provided through parameters. We suppose the word labels for options and subtasks are provided in **Cooking**. I.e. the words for parameters "pickup", "apple", etc. are known. Then, we can infer low level (but semantically meaningful) features from these words by using *word embeddings* to encode the

Ground Truth Attribute	Accuracy (on unseen test entities)
`isfood`	95%
`needslice`	90%
`iscookware`	70%
`isplace`	75%
`isboard`	95%

Table 7.1: We evaluate the generalization accuracy of PSGI on unseen test entities in the **Cooking** environment. For each ground truth attribute, we evaluate whether PSGI accurately labels the unseen test entity correctly. There are 10 seen entities (training set), and 20 unseen entities (test set) in the **Cooking** environment.

parameters Pennington et al. [2014]. We choose to use 50 dimensional GloVE word embeddings from Pennington et al. [2014]. We then evaluate by measuring the accuracy of attributes for **20** additional unseen test parameters, all words related to kitchens and cooking. We show the results in Table 7.1. From these results, we can extrapolate that at least **70%** of edges (on unseen entities) in the predicate subtask graph using these attributes are accurate.

7.6 Discussion

In this work we presented *factored subtask graph inference* (FSGI), a method for efficiently inferring the latent structure of hierarchical and compositional tasks. FSGI also facilitates inference of *unseen* subtasks during adaptation, by inferring relations using predicates. FSGI additionally learns *parameter attributes* in a zero-shot manner, which differentiate the structures of different predicate subtasks. Our experimental results showed that FSGI is more efficient and more general than prior work. In future work, we aim to to tackle noisy settings, where options and subtasks exhibit possible failures, and settings where the option policies must also be learned.

CHAPTER 8

Discussion and Future Work

In this thesis, I have proposed a framework that incorporates the inductive logic model into a deep reinforcement learning algorithm which enables sample-efficient learning and strong generalization. I first formulated the subtask graph framework which formally defines the compositional structure in the task in terms of the subtasks and their dependency, which is the precondition. Specifically, we modeled the completion, eligibility, and reward of each subtask and defined the subtask graph that characterizes all the subtasks in the task. Then, I developed a deep neural network-based reinforcement learning model (*i.e.*, NSGS) that can understand and solve any given (unseen) task which can be represented as a subtask graph. Next, I extended this work to a few-shot reinforcement learning setting where the agent is *not* given the subtask graph input. I proposed to incorporate the inductive logic programming method for inferring the latent task structure from the agent's trajectory, which achieved a strong few-shot reinforcement learning performance in many challenging domains such as StarCraft II. Subsequently, this work has been extended in two directions to further improve the efficiency via modeling the prior and factored structure. For modeling the prior, I proposed the policy mixing strategy for both adaptation and test policies which has been shown to be a simple yet effective method to model the prior of multiple similar subtask graphs. Its effectiveness is demonstrated on the web navigations tasks where the agent should execute a similar task (*e.g.*, placing a shipping order) on different websites. For modeling the factored structure, I proposed to model each subtask in terms of the entities and looked for the factored form of the subtask graph where each factor can be replaced with a certain set of entities. I showed that such factored form compactly captures the task structure and enables the knowledge sharing among similar subtasks and achieves a stronger form of generalization toward unseen subtasks with unseen entities.

The proposed methods have strong benefits over the existing multi-task/meta-reinforcement learning algorithms in terms of sample efficiency. This is due to the deterministic learning nature of the logic induction module, which can in principle build a perfect prediction model for the data after observing it only once. However, the subtask graph framework is limited in applicability due to several assumptions it makes on the task structure and the environment. Though my efforts

to develop more general subtask graph frameworks in my thesis, there are still many remaining limitations that call for future work:

Noise in the observation:

The inductive logic algorithm assumes that the input data has no error. Since our task inference module adopts the logic induction method, it is also limited to the environment in which states or observation has no noise. In a real-world application, however, the observation inevitably involves the sensor noise. Thus, it is crucial to develop an inductive logic algorithm robust to the input data noise. In principle, the algorithm presented in Chapter 5 can handle the noisy state input. However, it is designed mainly for handling the different subtask set between two arbitrary tasks, which may not necessarily be the best model for handling noisy state input. To this end, the first research problem to be tackled is to formally model the "noise" in the state space in the subtask graph framework. From the formulated noise model, it would be an interesting direction to extend the current decision tree-based logic induction method to a graph neural network-based model which learns to correct the input data error from statistics.

Completion and eligibility of the subtask:

In all the works presented in this thesis, it is assumed that the completion (*i.e.*, whether a subtask is complete) and the eligibility (*i.e.*, whether the precondition of a subtask is satisfied) of each subtask are known to the agent. However, in a real-world scenario, such high-level information may not be readily available. Thus, it would be an important and interesting direction to study whether the agent can "discover" the subtasks and their completion and eligibility predictor from a raw observation. In fact, from the connection between the subtask graph framework and the options framework (See Chapter 3 and Chapter 4 for more detail), discovering the subtasks from the agent's trajectory can be seen as the hierarchical imitation learning problem [Le et al., 2018, Shiarlis et al., 2018, Kipf et al., 2019, Gupta et al., 2019, Mandlekar et al., 2020]. The main difference between the subtask graph framework and the options framework is the assumption that the precondition can be defined in terms of the completion of other subtasks, which should be incorporated into the existing hierarchical imitation learning algorithms. Considering the recent advances [Kipf et al., 2019, Mandlekar et al., 2020] in hierarchical imitation learning methods, I foresee that the subtask can be discovered from the agent's observation in a self-supervised manner from the temporal and visual saliency information.

APPENDIX A

Multi-task Reinforcement Learning for Compositional Task with Given Task Description

A.1 Details of the Task Parameterized by Subtask Graph

We define each task as an MDP tuple $\mathcal{M}_G = (\mathcal{S}, \mathcal{A}, \mathcal{P}_G, \mathcal{R}_G, \rho_G, \gamma)$ where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{P}_G : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is a task-specific state transition function, $\mathcal{R}_G : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a task-specific reward function and $\rho_G : \mathcal{S} \to [0, 1]$ is a task-specific initial distribution over states. We describe the subtask graph G and each component of MDP in the following paragraphs.

Subtask and Subtask Graph The subtask graph consists of N subtasks that is a subset of $\mathcal{O}$, the subtask reward $\mathbf{r} \in \mathbb{R}^N$, and the precondition of each subtask. The set of subtasks is $\mathcal{O} = \mathcal{A}_{int} \times \mathcal{X}$, where $\mathcal{A}_{int}$ is a set of primitive actions to interact with objects, and $\mathcal{X}$ is a set of all types of interactive objects in the domain. To execute a subtask $(a_{int}, obj) \in \mathcal{A}_{int} \times \mathcal{X}$, the agent should move on to the target object obj and take the primitive action a_{int}.

State The state $\mathbf{s}_t$ consists of the observation $\mathbf{obs}_t \in \{0, 1\}^{W \times H \times C}$, the completion vector $\mathbf{x}_t \in \{0, 1\}^N$, the time budget $step_t$ and the eligibility vector $\mathbf{e}_t \in \{0, 1\}^N$. An observation $\mathbf{obs}_t$ is represented as $H \times W \times C$ tensor, where H and W are the height and width of map respectively, and C is the number of object types in the domain. The (h, w, c)-th element of observation tensor is 1 if there is an object c in (h, w) on the map, and 0 otherwise. The time budget indicates the number of remaining time-steps until the episode termination. The completion vector and eligibility vector provides additional information about N subtasks. The details of completion vector and eligibility vector will be explained in the following paragraph.

State Distribution and Transition Function Given the current state $(\mathbf{obs}_t, \mathbf{x}_t, \mathbf{e}_t)$, the next step state $(\mathbf{obs}_{t+1}, \mathbf{x}_{t+1}, \mathbf{e}_{t+1})$ is computed from the subtask graph G. In the beginning of episode, the

initial time budget $step_t$ is sampled from a pre-specified range N_{step} for each subtask graph (See section A.3 for detail), the completion vector $\mathbf{x}_t$ is initialized to a zero vector in the beginning of the episode $\mathbf{x}_0 = [0, \ldots, 0]$ and the observation $\mathbf{obs}_0$ is sampled from the task-specific initial state distribution ρ_G. Specifically, the observation is generated by randomly placing the agent and the N objects corresponding to the N subtasks defined in the subtask graph G. When the agent executes subtask i, the i-th element of completion vector is updated by the following update rule:

$$x_{t+1}^i = \begin{cases} 1 & \text{if} \quad e_t^i = 1 \\ x_t^i & \text{otherwise} \end{cases}. \tag{A.1}$$

The observation is updated such that agent moves on to the target object, and perform corresnponding primitive action (See Appendix A.2 for the full list of subtasks and corresponding primitive actions on Mining and Playground domain). The eligibility vector $\mathbf{e}_{t+1}$ is computed from the completion vector $\mathbf{x}_{t+1}$ and subtask graph G as follows:

$$e_{t+1}^i = \underset{j \in Child_i}{\text{OR}} \left(y_{AND}^j \right), \tag{A.2}$$

$$y_{AND}^i = \underset{j \in Child_i}{\text{AND}} \left(\hat{x}_{t+1}^{i,j} \right), \tag{A.3}$$

$$\hat{x}_{t+1}^{i,j} = x_{t+1}^j w^{i,j} + (1 - x_{t+1}^j)(1 - w^{i,j}), \tag{A.4}$$

where $w^{i,j} = 0$ if there is a `NOT` connection between i-th node and j-th node, otherwise $w^{i,j} = 1$. Intuitively, $\hat{x}_t^{i,j} = 1$ when j-th node does not violate the precondition of i-th node. Executing each subtask costs different amount of time depending on the map configuration. Specifically, the time cost is given as the Manhattan distance between agent location and target object location in the grid-world plus one more step for performing a primitive action.

Task-specific Reward Function The reward function is defined in terms of the subtask reward vector $\mathbf{r}$ and the eligibility vector $\mathbf{e}_t$, where the subtask reward vector $\mathbf{r}$ is the component of subtask graph G the and eligibility vector is computed from the completion vector $\mathbf{x}_t$ and subtask graph G as Eq. A.4. Specifically, when agent executes subtask i, the reward given to agent at time step t is given as follows:

$$r_t = \begin{cases} r^i & \text{if} \quad e_t^i = 1 \\ 0 & \text{otherwise} \end{cases}. \tag{A.5}$$

A.2 Details of Playground and Mining Domains

A.2.1 Mining

There are 15 types of objects: *Mountain*, *Water*, *Work space*, *Furnace*, *Tree*, *Stone*, *Grass*, *Pig*, *Coal*, *Iron*, *Silver*, *Gold*, *Diamond*, *Jeweler's shop*, and *Lumber shop*. The agent can take 10 primitive

actions: *up*, *down*, *left*, *right*, *pickup*, *use1*, *use2*, *use3*, *use4*, *use5* and agent cannot moves on to the *Mountain* and *Water* cell. *Pickup* removes the object under the agent, and *use*'s do not change the observation. There are 26 subtasks in the Mining domain:

- Get wood/stone/string/pork/coal/iron/silver/gold/diamond: The agent should go to *Tree*/*Stone*/*Grass*/*Pig*/*Coal*/*Iron*/*Silver*/*Gold*/*Diamond* respectively, and take *pickup* action.
- Make firewood/stick/arrow/bow: The agent should go to *Lumber shop* and take *use1*/*use2*/*use3*/*use4* action respectively.
- Light furnace: The agent should go to *Furnace* and take *use1* action.
- Smelt iron/silver/gold: The agent should go to *Furnace* and take *use2*/*use3*/*use4* action respectively.
- Make stone-pickaxe/iron-pickaxe/silverware/goldware/bracelet: The agent should go to *Work space* and take *use1*/*use2*/*use3*/*use4*/*use5* action respectively.
- Make earrings/ring/necklace: The agent should go to *Jeweler's shop* and take *use1*/*use2*/*use3* action respectively.

The icons used in Mining domain were downloaded from `www.icons8.com` and `www.flaticon.com`. The *Diamond* and *Furnace* icons were made by Freepik from `www.flaticon.com`.

A.2.2 Playground

There are 10 types of objects: *Cow*, *Milk*, *Duck*, *Egg*, *Diamond*, *Heart*, *Box*, *Meat*, *Block*, and *Ice*. The *Cow* and *Duck* move by 1 pixel in random direction with the probability of 0.1 and 0.2, respectively. The agent can take 6 primitive actions: *up*, *down*, *left*, *right*, *pickup*, *transform* and agent cannot moves on to the *block* cell. *Pickup* removes the object under the agent, and *transform* changes the object under the agent to *Ice*. The subtask graph was randomly generated without any hand-coded template (see Section A.3 for details).

A.3 Details of Subtask Graph Generation on Playground and Mining Domains

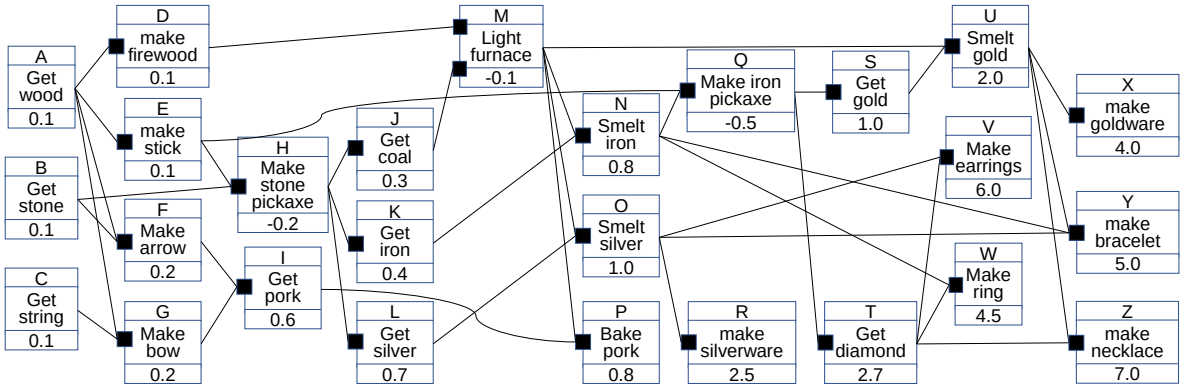


Figure A.1: The entire graph of Mining domain. Based on this graph, we generated 640 subtask graphs by removing the subtask node that has no parent node.

A.3.1 Mining Domain

The precondition of each subtask in Mining domain was defined as Figure A.1. Based on this graph, we generated all possible sub-graphs of it by removing the subtask node that has no parent node, while always keeping subtasks A, B, D, E, F, G, H, I, K, L. The reward of each subtask was randomly scaled by a factor of $0.8 \sim 1.2$.

A.3.2 Playground Domain

Nodes	N_T	number of tasks in each layer
	N_D	number of distractors in each layer
	N_A	number of `AND` node in each layer
	r	reward of subtasks in each layer
Edges	N_{ac}^{+}	number of children of `AND` node in each layer
	N_{ac}^{-}	number of children of `AND` node with `NOT` connection in each layer
	N_{dp}	number of parents with `NOT` connection of distractors in each layer
	N_{oc}	number of children of `OR` node in each layer
Episode	N_{step}	number of step given for each episode

Table A.1: Parameters for generating task including subtask graph parameter and episode length.

For training and test sample generation, the subtask graph structure was defined in terms of the parameters in table A.1. To cover wide range of subtask graphs, we randomly sampled the parameters $N_A, N_O, N_{ac}^{+}, N_{ac}^{-}, N_{dc}$, and N_{oc} from the range specified in the table A.2 and A.3, while N_T and N_D was manually set. We prevented the graph from including the duplicated `AND` nodes with the same children node(s). We carefully set the range of each parameter such that at least 500 different subtask graphs can be generated with the given parameter ranges. The table A.2

summarizes parameters used to generate training and evaluation subtask graphs for the Playground domain.

Train (=**D1**)	N_T	{6,4,2,1}
	N_D	{2,1,0,0}
	N_A	{3,3,2}-{5,4,2}
	N_{ac}^+	{1,1,1}-{3,3,3}
	N_{ac}^-	{0,0,0}-{2,2,1}
	N_{dp}	{0,0,0}-{3,3,0}
	N_{oc}	{1,1,1}-{2,2,2}
	r	{0.1,0.3,0.7,1.8}-{0.2,0.4,0.9,2.0}
	N_{step}	48-72
D2	N_T	{7,5,2,1}
	N_D	{2,2,0,0}
	N_A	{4,3,2}-{5,4,2}
	N_{ac}^+	{1,1,1}-{3,3,3}
	N_{ac}^-	{0,0,0}-{2,2,1}
	N_{dp}	{0,0,0,0}-{3,3,0,0}
	N_{oc}	{1,1,1}-{2,2,2}
	r	{0.1,0.3,0.7,1.8}-{0.2,0.4,0.9,2.0}
	N_{step}	52-78
D3	N_T	{5,4,4,2,1}
	N_D	{1,1,1,0,0}
	N_A	{3,3,3,2}-{5,4,4,2}
	N_{ac}^+	{1,1,1,1}-{3,3,3,3}
	N_{ac}^-	{0,0,0,0}-{2,2,1,1}
	N_{dp}	{0,0,0,0,0}-{3,3,3,0,0}
	N_{oc}	{1,1,1,1}-{2,2,2,2}
	r	{0.1,0.3,0.6,1.0,2.0}-{0.2,0.4,0.7,1.2,2.2}
	N_{step}	56-84
D4	N_T	{4,3,3,3,2,1}
	N_D	{0,0,0,0,0,0}
	N_A	{3,3,3,3,2}-{5,4,4,4,2}
	N_{ac}^+	{1,1,1,1,1}-{3,3,3,3,3}
	N_{ac}^-	{0,0,0,0,0}-{2,2,1,1,0}
	N_{dp}	{0,0,0,0,0,0}-{0,0,0,0,0,0}
	N_{oc}	{1,1,1,1,1}-{2,2,2,2,2}
	r	{0.1,0.3,0.6,1.0,1.4,2.4}-{0.2,0.4,0.7,1.2,1.6,2.6}
	N_{step}	56-84

Table A.2: Subtask graph parameters for training set and tasks **D1**∼**D4**.

Base	N_T	{4,3,2,1}
	N_D	{0,0,0,0}
	N_A	{3,3,2}-{4,3,3}
	N_{ac}^+	{1,1,2}-{3,2,2}
	N_{ac}^-	{0,0,0}-{0,0,0}
	N_{dp}	{0,0,0,0}-{0,0,0,0}
	N_{oc}	{1,1,1}-{2,2,2}
	N_{step}	40-60
-OR	N_{oc}	{1,1,1}-{1,1,1}
+Distractor	N_D	{2,1,0,0}
+NOT	N_{ac}^+	{0,0,0}-{3,2,2}
+NegDistractor	N_D	{2,1,0,0}
	N_{dp}	{0,0,0,0}-{3,3,0,0}
+Delayed	r	{0,0,0,1.6}-{0,0,0,1.8}

Table A.3: Subtask graph parameters for analysis of subtask graph components.

APPENDIX B

Meta Reinforcement Learning for Compositional Task via Task Inference

B.1 Details of the SC2LE Domain

The **SC2LE** domain [Vinyals et al., 2017] provides suite of mini-games focusing on specific aspects of the entire StarCraft II game. In this paper, we custom design two types of new, simple mini-games called *Build Unit* and *Defeat Zerg troops*. Specifically, we built *Defeat Zerglings, Defeat Hydralisks, Defeat Hydralisks & Ultralisks* and *Build Battlecruiser* mini-games that compactly capture the most fundamental goal of the full game. The *Build Unit* mini-game requires the agent to figure-out the target unit and its precondition correctly, such that it can train the target unit within the given short time budget. The *Defeat Zerg troops* mini-game mimics the full game more closely; the agent is required to train enough units to win a war against the opponent players. To make the task more challenging and interesting, we designed the reward to be given only at the end of episode depending on the success of the whole task. Similar to the standard *Melee* game in StarCraft II, each episode is initialized with 50 *mineral*, 0 *gas*, 7 and 4 *SCV*s that start gathering mineral and gas, respectively, 1 idle *SCV*, 1 refinery, and 1 *Command Center* (See Figure B.1). The episode is terminated after 2,400 environment steps (equivalent to 20 minutes in game time). In the game, the agent is initially given 50 *mineral*, 0 *gas*, 7 and 4 *SCV*s that start gathering mineral and gas, respectively, 1 idle *SCV*, 1 refinery, and 1 *Command Center* (See Figure B.1) and is allowed to prepare for the upcoming battle only for 2,400 environment steps (equivalent to 20 minutes in game time). Therefore, the agent must learn to collect resources and efficiently use them to build structures for training units. All the four custom mini-games share the same initial setup as specified in Figure B.1.

***Defeat Zerg troops* scenario**: At the end of the war preparation, different combinations of enemy unit appears: *Defeat Zerglings* and *Defeat Hydralisks* has 20 zerglings and 15 hydralisks, respectively, and *Defeat Hydralisks & Ultralisks* contains a combination of total 5 hydralisks and 3 ultralisks. When the war finally breaks out, the units trained by the agent will encounter the army of Zerg units in the map and combat until the time over (240 environment steps or 2 minutes in

the game) or either side is defeated. Specifically, the agent may not take any action, and the units

Figure B.1: (**Top**) The agent starts the game initially with limited resources of 50 minerals, 0 gases, 3 foods, 11 SCVs collecting resources, 1 idle SCV and pre-built Refinery. (**Middle**) From the initial state, the agent needs to strategically collect resources and build structures in order to be well prepared for the upcoming battle. (**Bottom**) After 2,400 environment steps, the war breaks; all the buildings in the map are removed, and the enemy units appear. The agent's units should eliminate the enemy units within 240 environment steps during the war.

trained by the agent perform an *auto attack* against the enemy units. Unlike the original full game that has ternary reward structure of $+1$ (win) / 0 (draw) / -1 (loss), we use binary reward structure of $+1$ (win) and -1 (loss or draw). Notice that depending on the type of units the agent trained, a draw can happen. For instance, if the units trained by the agent are air units that cannot attack the ground units and the enemy units are the ground units that cannot attack the air units, then no combat will take place, so we consider this case as a loss. ***Build unit* scenario**: The agent receives the reward of $+1$ if the target unit is successfully trained within the time limit, and the episode terminates. When the episode terminates due to time limit, the agent receives the reward of -1. We gave 2,400 step budget for the *Build Battlecruiser* scenario such that only highly efficient policy can finish the task within the time limit.

The transition dynamics (*i.e.*, build tech-tree) in **SC2LE** domain has a hierarchical characteristic which can be inferred by our MSGI agent (see Figure B.1). We conducted the experiment on Terran race only, but our method can be applied to other races as well.

Subtask. There are 85 subtasks: 15 subtasks of constructing each type of building (*Supply depot, Barracks, Engineeringbay, Refinery, Factory, Missile turret, Sensor tower, Bunker, Ghost academy, Armory, Starport, Fusioncore, Barrack-techlab, Factory-techlab, Starport-techlab*), 17 subtasks of training each type of unit (*SCV, Marine, Reaper, Marauder, Ghost, Widowmine, Hellion, Hellbat, Cyclone, Siegetank, Thor, Banshee, Liberator, Medivac, Viking, Raven, Battlecruiser*), one subtask of idle worker, 32 subtasks of selecting each type of building and unit, gathering mineral, gathering gas, and no-op. For gathering mineral, we set the subtask as (mineral$\geq val$) where $val \in \{50, 75, 100, 125, 150, 300, 400\}$. Similarly for gathering gas, we set the subtask as (gas$\geq val$) where $val \in \{25, 50, 75, 100, 125, 150, 200, 300\}$. For no-op subtask, the agent takes the no-op action for 8 times.

Eligibility. The eligibility of the 15 building construction subtasks and 17 training unit subtasks is given by the environment as an *available action* input. For the selection subtasks, we extracted the number of corresponding units using the provided API of the environment. Gathering mineral, gas, and no-op subtasks are always eligible.

Completion. The completion of the 15 construction subtasks and 17 training subtasks is 1 if the corresponding building or unit is present on the map. For the selection subtasks, the completion is 1 if the target building or unit is selected. For gathering mineral and gas subtasks, the subtask is completed if the condition is satisfied (*i.e.*, gas≥ 50). The no-op subtask is never completed.

Subtask reward. In SC2LE domain, the agent does not receive any reward when completing a subtask. The only reward given to agent is the binary reward $r_{H_{\text{epi}}} = \{+1, -1\}$ at the end of episode (*i.e.*, $t = H_{\text{epi}}$). Therefore, the subtask reward inference method described in Eq.(4.3) may not be applied. In the adaptation phase, we used the same UCB-inspired exploration bonus, introduced in Section 4.4.3 with GRProp policy. In test phase, we simply used $+1.0$ subtask reward for all the

unit production subtasks, and run our MSGI-GRProp agent in test phase.

APPENDIX C

Fast Inference and Transfer of Compositional Task Structures for Few-shot Task Generalization

C.1 Details on SymWoB domain

In this paper, we introduce the **SymWoB** domain, which is a challenging symbolic environment that aims to reflect the hierarchical and compositional aspects of the checkout processes in the real-world websites. There are total 10 different **SymWoB** websites that are symbolic implementations of the actual websites: **Amazon**, **Apple**, **Dick's**, **Walmart**, **Converse**, **Target**, **eBay**, **Ikea**, **BestBuy**, and **Samsung**. The main goal of each website is to navigate through the web pages within the website by clicking and filling in the web elements with proper information which leads to the final web page that allows the agent to click on the `Place_Order` button, which indicates that the agent has successfully finished the task of checking out.

C.1.1 Implementation detail

In this section, we describe the detailed process of implementing an existing website into a symbolic version. We first fill out the shopping cart with random products, and we proceed until placing the order on the actual website. During the process, we extract all the interactable web elements on the webpage. We repeat this for all the websites and form a shared subtask pool where similar web elements in different websites that have same functionality are mapped to the same subtask in the shared subtask pool. Then, we extract the precondition relationship between subtasks from the website and form the edges in the subtask graph accordingly. Finally, we implement the termination condition and the subtask reward to the failure distractor (See Appendix C.1.3) and the goal subtasks.

C.1.2 Comparison of the websites

The agent's goal on every website is the same (*i.e.*placing a checkout order), but the underlying subtask graphs, or task structure, of the websites are extremely diverse, making the task much more challenging for the agent. One of the major sources of diversity in subtask graphs is in the various ordering of the web pages. In a typical website's checkout process, some of the most common web pages include the shipping, billing, and payment web pages, each of which has a collection of corresponding subtasks. In Figure C.1, for example, the shipping web page is represented by the collection of the subtasks on the left side from `Fill_Zip` to `Fill_Last` and `Click_ContinueShipping`, and these come *before* the payment web page that is represented by the subtasks on the right side from `Click_Credit` to `Click_ContinuePayment`. On the other hand, in Figure C.7 and Figure C.6, the similar web pages are either connected in a different ordering, from payment to shipping web page, or placed on the same line side by side. Since the web pages can vary on how they are ordered, it allows the subtask graphs to have a variety of shapes such as deep and narrow as in Figure C.9 or wide and shallow as in Figure C.6. Different shape of the subtask graphs means different precondition between the tasks, making it non-trivial for the agent to transfer its knowledge about one to the other.

Another major source of diversity is the number of web elements in each web page. Let's compare the web elements of the shipping web page in Figure C.3 and Figure C.4. These are the subtasks that are connected to `Click_ContinueShipping` and as well as itself. We can see that the two websites do not have the same number of the web elements for the shipping web pages: the **Converse** website requires more shipping information to be filled out than the **Dick's** website. Such variety in the number of web elements, or subtasks, allows the subtask graphs of the websites to have diverse preconditions as well.

C.1.3 Distractor subtasks

In addition to the different task structures among the websites, there are also *distractor* subtasks in the websites that introduces challenging components of navigating the real-world websites. There are two different types of distractor subtasks: the one that terminates the current episode with a negative reward and the another one that has no effect. The former, which we also call it the *failure* distractor subtask, represents the web elements that lead the agent to some external web pages like `Help` or `Terms_of_Use` button. The latter is just called the distractor subtask, where executing the subtask does not contribute to progressing toward the goal (*e.g.*, `Click EditShipping` subtask in **Converse**). Each website has varying number of distractor subtasks and along with the shallowness of the task structure, the number of distractor subtasks significantly affects the difficulty of the task.

C.1.4 Environment generation

All of the websites in the **SymWoB** domain are generated by analyzing the corresponding real websites and reflecting their key aspects of checkout process.

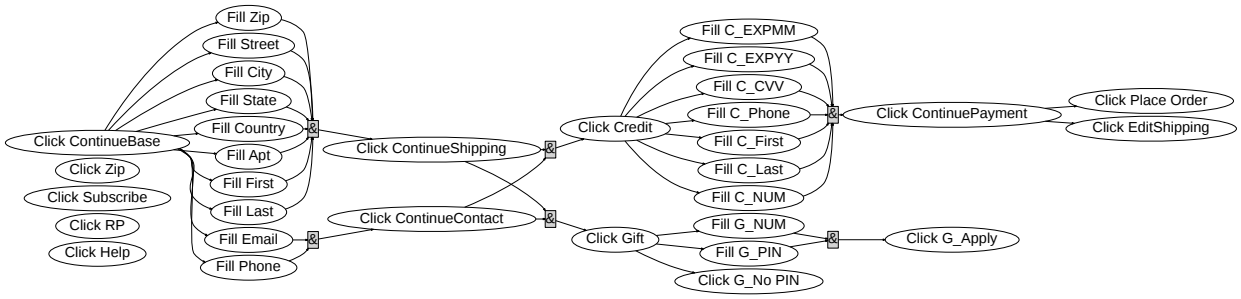


Figure C.1: The ground-truth subtask graph of **Walmart** website.

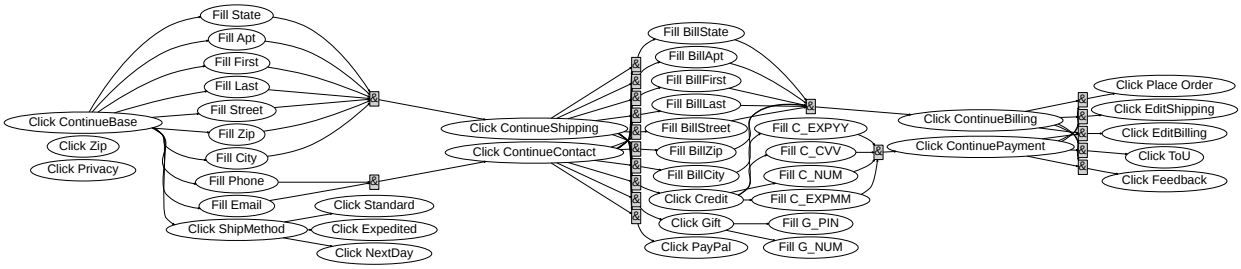


Figure C.2: The ground-truth subtask graph of **Converse** website.

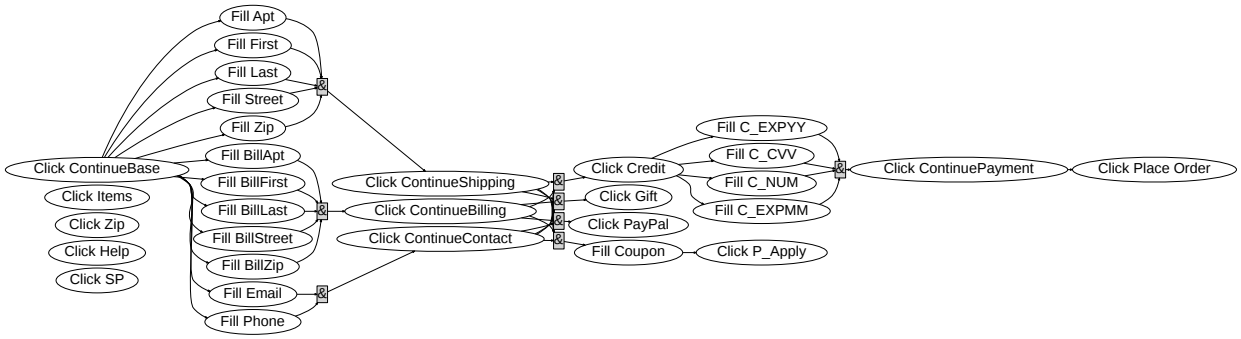


Figure C.3: The ground-truth subtask graph of **Dick's** website.

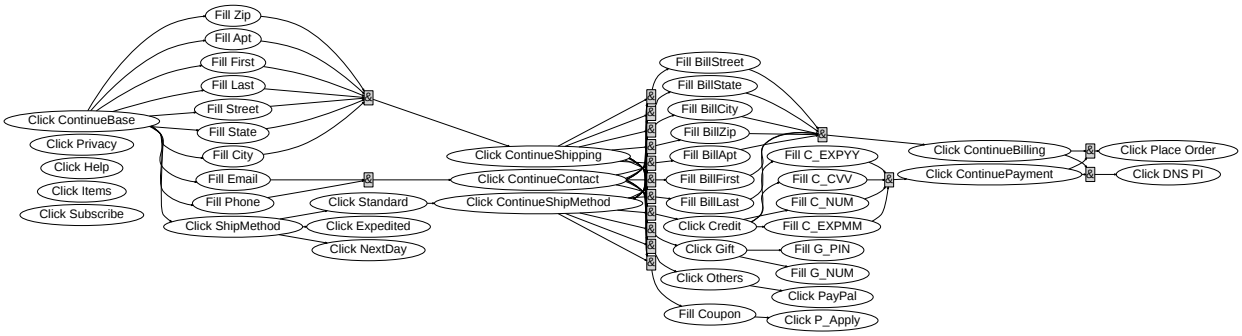


Figure C.4: The ground-truth subtask graph of **BestBuy** website.

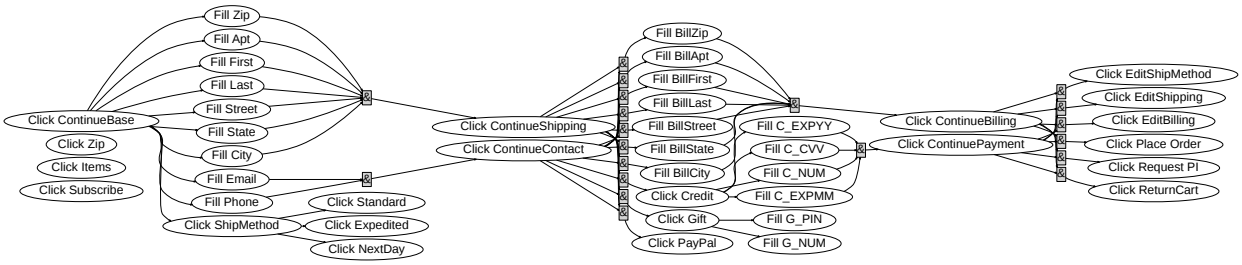


Figure C.5: The ground-truth subtask graph of **Apple** website.

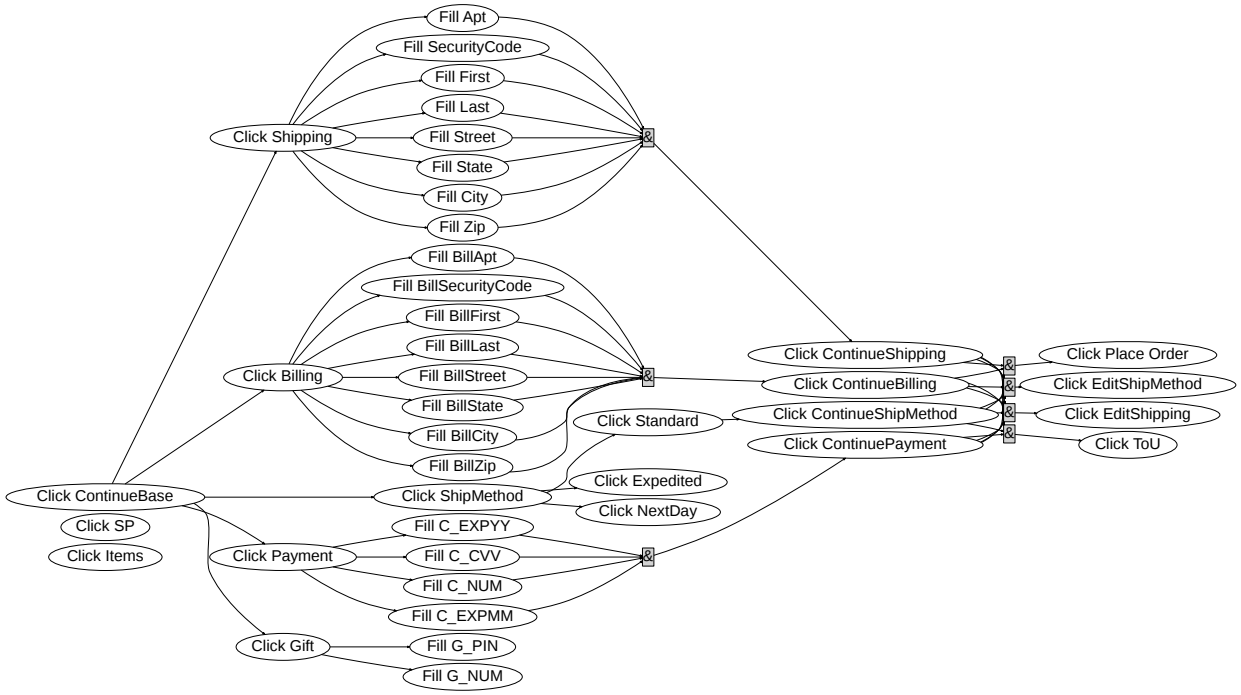


Figure C.6: The ground-truth subtask graph of **Amazon** website.

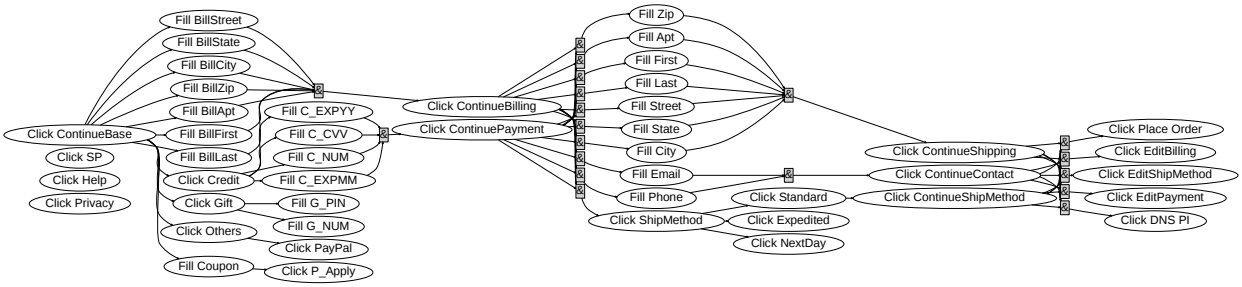


Figure C.7: The ground-truth subtask graph of **Samsung** website.

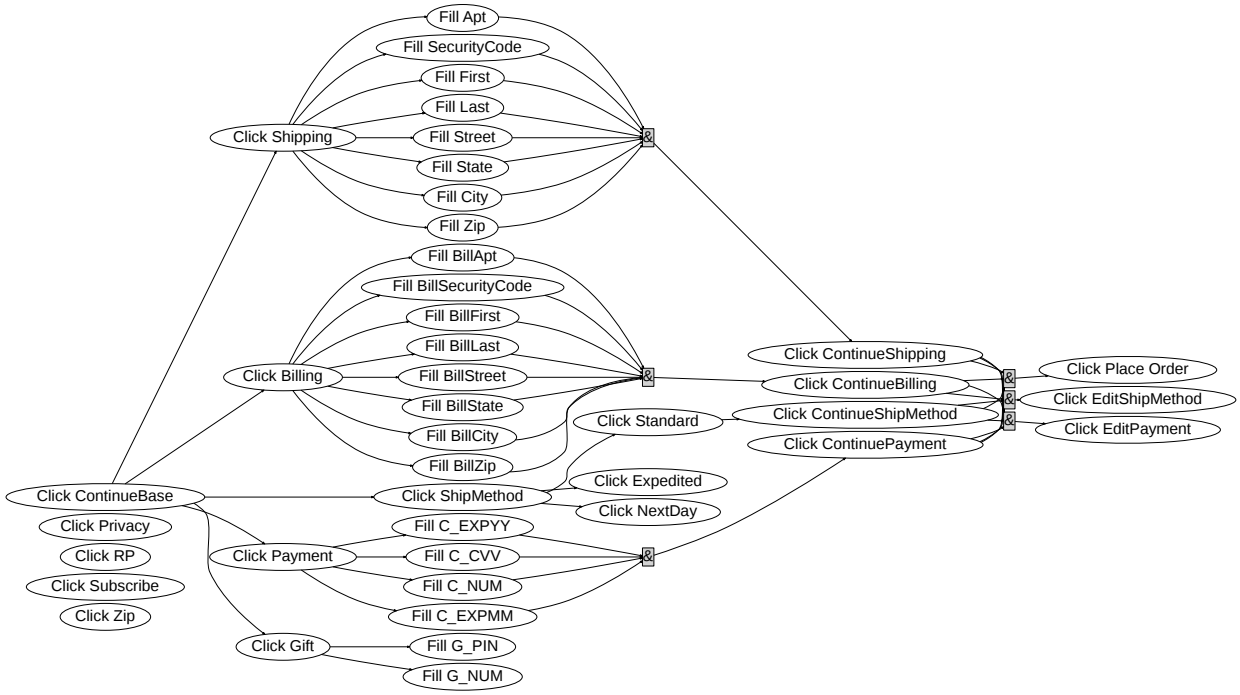


Figure C.8: The ground-truth subtask graph of **eBay** website.

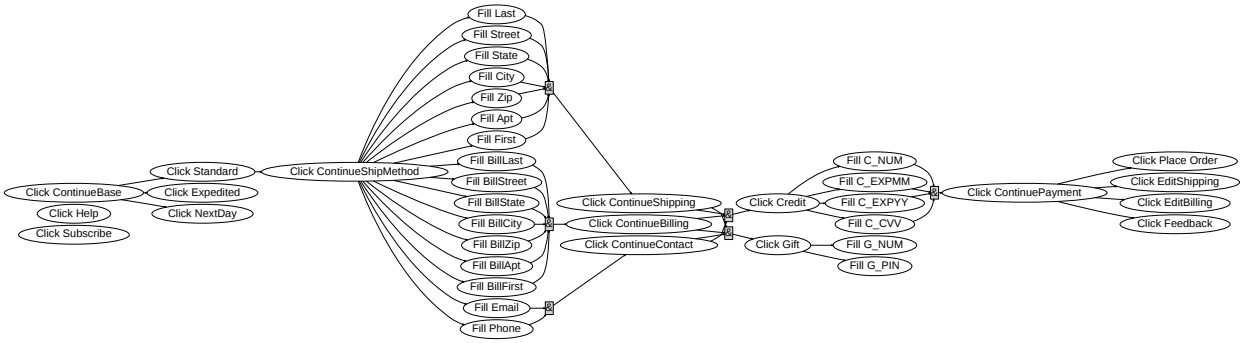


Figure C.9: The ground-truth subtask graph of **Ikea** website.

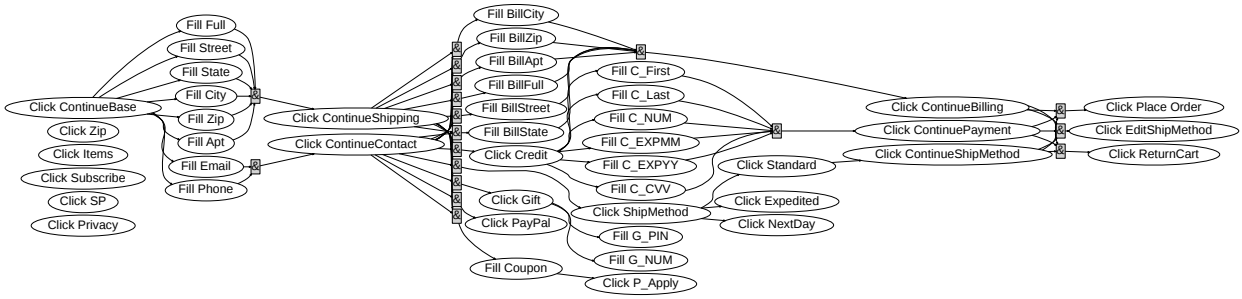


Figure C.10: The ground-truth subtask graph of **Target** domain.

APPENDIX D

Shortest-Path Constrained Reinforcement Learning for Sparse Reward Tasks

D.1 Option framework-based formulation

D.1.1 Preliminary: option framework

Options framework [Sutton, 1998] defines options as a generalization of actions to include temporally extended series of action. Formally, options consist of three components: a policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, a termination condition $\beta : \mathcal{S}^+ \to [0, 1]$, and an initiation set $\mathcal{I} \subseteq \mathcal{S}$. An option $\langle \mathcal{I}, \pi, \beta \rangle$ is available in state s if and only if $s \in \mathcal{I}$. If the option is taken, then actions are selected according to π until the option terminates stochastically according to β. Then, the option-reward and option-transition models are defined as

$$r_s^o = \mathbb{E}\left\{ r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{k-1} r_{t+k} \mid E(o, s, t) \right\} \tag{D.1}$$

$$P_{ss'}^o = \sum_{k=1}^{\infty} p\left(s', k\right) \gamma^k \tag{D.2}$$

where $t + k$ is the random time at which option o terminates, $E(o, s, t)$ is the event that option o is initiated in state s at time t, and $p(s', k)$ is the probability that the option terminates in s' after k steps. Using the option models, we can re-write Bellman equation as follows:

$$V^\pi(s) = \mathbb{E}\left[r_{t+1} + \cdots + \gamma^{k-1} r_{t+k} + \gamma^k V^\pi(s_{t+k}) \right], \tag{D.3}$$

$$= \sum_{o \in \mathcal{O}} Pr[E(o, s)] \left[r_s^o + \sum_{s'} P_{ss'}^o V^\pi(s') \right]. \tag{D.4}$$

where $t + k$ is the random time at which option o terminates and $E(o, s)$ is the event that option o is initiated in state s.

D.1.2 Option-based view-point of shortest-path constraint

In this section, we present an option framework-based viewpoint of our shortest-path (SP) constraint. We will first show that a (sparse-reward) MDP can be represented as a weighted directed graph where nodes are rewarding states, and edges are options. Then, we show that a policy satisfying SP constraint also maximizes the option-transition probability $P^o_{ss'}$.

For a given MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \rho, \bar{\mathcal{S}})$, let $\mathcal{S}^R = \{s | R(s) \neq 0\} \subset \mathcal{S}$ be the set of all rewarding states, where $R(s)$ is the reward function upon arrival to state s. In sparse-reward tasks, it is assumed that $|\mathcal{S}^R| << |\mathcal{S}|$. Then, we can form a weighted directed graph $G^\pi = (\mathcal{V}, \mathcal{E})$ of policy π and given MDP. The vertex set is defined as $\mathcal{V} = \mathcal{S}^R \cup \rho_0 \cup \bar{\mathcal{S}}$ where $\mathcal{S}^R$ is rewarding states, ρ_0 is the initial states, and $\bar{\mathcal{S}}$ is the terminal states. Similar to the path set in Definition 2, let $\mathcal{T}_{s \to s'}$ denotes a set of paths transitioning from one vertex $s \in \mathcal{V}$ to another vertex $s' \in \mathcal{V}$:

$$\mathcal{T}_{s \to s'} = \{\tau | s_0 = s, s_{\ell(\tau)} = s', \{s_t\}_{0<t<\ell(\tau)} \cap \mathcal{V} = \emptyset\}. \tag{D.5}$$

Then, the edge from a vertex $s \in \mathcal{V}$ to another vertex $s' \in \mathcal{V}$ is defined by an (implicit) option tuple: $o(s, s') = (\mathcal{I}, \pi, \beta)_{(s,s')}$, where $\mathcal{I} = \{s\}$, $\beta(s) = \mathbb{I}(s = s')$, and

$$\pi^{(s,s')}(\tau) = \begin{cases} \frac{1}{Z}\pi(\tau) & \text{for } \tau \in \mathcal{T}_{s \to s'} \\ 0 & \text{otherwise} \end{cases}, \tag{D.6}$$

where Z is the partition function to ensure $\int \pi^{(s,s')}(\tau) d\tau = 1$. Following Equation (D.1), the option-reward is given as

$$r^\pi_{s,s'} = \mathbb{E}^{\pi^{(s,s')}} \left[r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{k-1} r_{t+k} \mid E(o^{(s,s')}, s, t) \right], \tag{D.7}$$

$$= \mathbb{E}^{\pi^{(s,s')}} \left[\gamma^{k-1} r_{t+k} \mid E(o^{(s,s')}, s, t) \right], \tag{D.8}$$

where $t + k$ is the random time at which option $o(s, s')$ terminates, and $E(o, s, t)$ is the event that option $o(s, s')$ is initiated in state s at time t. Note that in the last equality, $r_{t+1} = \cdots = r_{t+k-1} = 0$ holds since $\{s_{t+1}, \ldots, s_{t+k-1}\} \cap \mathcal{V} = \emptyset$ from the definition of option policy $\pi^{(s,s')}$. Following Equation (D.2), the option transition is given as

$$P^\pi_{s,s'} = \sum_{k=1}^{\infty} p(s', k) \gamma^k \tag{D.9}$$

$$= \mathbb{E}^\pi \left[\gamma^k | s_0 = s, s_k = s', \{r_t\}_{t<k} = 0 \right] \tag{D.10}$$

$$= \gamma^{D^\pi_{\text{nr}}(s,s')}. \tag{D.11}$$

where $p(s', k)$ is the probability that the option terminates in s' after k steps, and $D^\pi_{\text{nr}}(s, s')$ is the π-distance in Definition 3. Then, we can re-write the shortest-path constraint in terms of $P^\pi_{s,s'}$ as

follows:

$$\Pi^{\text{SP}} = \{\pi | \forall (s, s' \in \mathcal{T}^{\pi}_{\hat{s},\hat{s}',\text{nr}} \text{ s.t. } (\hat{s}, \hat{s}') \in \Phi^{\pi}),\ D^{\pi}_{\text{nr}}(s, s') = \min_{\pi} D^{\pi}_{\text{nr}}(s, s')\} \tag{D.12}$$

$$= \{\pi | \forall (s, s' \in \mathcal{T}^{\pi}_{\hat{s},\hat{s}',\text{nr}} \text{ s.t. } (\hat{s}, \hat{s}') \in \Phi^{\pi}),\ P^{\pi}_{s,s'} = \max_{\pi} P^{\pi}_{s,s'}\} \tag{D.13}$$

Thus, we can see that the policy satisfying SP constraint also maximizes the option-transition probability. We will use this result in Appendix D.2.

D.2 Shortest-Path Constraint: A Single-goal Case

In this section, we provide more discussion on a special case of the shortest-path constraint (Section 6.3.1), when the (stochastic) MDP defines a single-goal task: *i.e.*, there exists a unique initial state $s_{\text{init}} \in \mathcal{S}$ and a unique goal state $s_g \in \mathcal{S}$ such that s_g is a terminal state, and $R(s) > 0$ if and only if $s = s_g$.

We first note that the non-rewarding path set is identical to the path set in such a setting, because the condition $r_t = 0 (t < \ell(\tau))$ from Definition 2 is always satisfied as $R(s) > 0 \Leftrightarrow s = s_g$ and $s_{\ell(\tau)} = s_g$:

$$\mathcal{T}^{\pi}_{s,s',\text{nr}} = \mathcal{T}^{\pi}_{s,s'} = \{\tau \mid s_0 = s, s_{\ell(\tau)} = s', p_{\pi}(\tau) > 0, \{s_t\}_{t<\ell(\tau)} \neq s'\} \tag{D.14}$$

Again, $\mathcal{T}^{\pi}_{s,s'}$ is a set of all path starting from s (*i.e.*, and ending at s' (*i.e.*, $s_{\ell(\tau)} = s'$) where the agent visits s' *only* at the end (*i.e.*, $\{s_t\}_{t<\ell(\tau)} \neq s'$), that can be rolled out by policy with a non-zero probability (*i.e.*, $p_{\pi}(\tau) > 0$).

We now claim that an optimal policy satisfies the shortest-path constraint. The idea is that, since s_g is the only rewarding and terminal state, maximizing $R(\tau) = \gamma^T R(s_g)$ where $s_T = s_g$ corresponds to minimizing the number of time steps T to reach s_g. In this setting, a shortest-path policy is indeed optimal.

Lemma 11. *For a single-goal MDP, any optimal policy satisfies the shortest-path constraint.*

Proof. Let s_{init} be the initial state and s_g be the goal state. We will prove that any optimal policy is a shortest-path policy from the initial state to the goal state. We use the fact that s_g is the only

rewarding state, *i.e.*, $R(s) > 0$ entails $s = s_g$.

$$\pi^* = \arg\max_\pi \mathbb{E}^{\tau\sim\pi}_{s\sim\rho}\left[\sum_t \gamma^t r_t \,\middle|\, s_0 = s\right] \tag{D.15}$$

$$= \arg\max_\pi \mathbb{E}^{\tau\sim\pi}\left[\sum_t \gamma^t r_t \,\middle|\, s_0 = s_{\text{init}}\right] \tag{D.16}$$

$$= \arg\max_\pi \mathbb{E}^{\tau\sim\pi}\left[\gamma^T R(s_g) \mid s_0 = s_{\text{init}}, s_{\ell(\tau)} = s_g\right] \tag{D.17}$$

$$= \arg\max_\pi \mathbb{E}^{\tau\sim\pi}\left[\gamma^T \mid s_0 = s_{\text{init}}, s_{\ell(\tau)} = s_g\right] \tag{D.18}$$

$$= \arg\min_\pi \log_\gamma\left(\mathbb{E}^{\tau\sim\pi}\left[\gamma^T \mid s_0 = s_{\text{init}}, s_{\ell(\tau)} = s_g\right]\right) \tag{D.19}$$

$$= \arg\min_\pi D^\pi_{\text{nr}}(s_{\text{init}}, s_g), \tag{D.20}$$

where Equation (D.18) holds since $R(s_g) > 0$ from our assumption that $R(s) + V^*(s) > 0$. □

D.3 Proof of Theorem 7

We make the following assumptions on the Markov Decision Process (MDP) $\mathcal{M}$: namely *mild stochasticity* (Definitions 12 and 13).

Definition 12 (Mild stochasticity (1))**.** In MDP $\mathcal{M}$, there exists an optimal policy π^* and the corresponding shortest-path policy $\pi^{sp} \in \Pi^{\text{SP}}$ such that for all $s, s' \in \Phi^\pi$, it holds $p_{\pi^*}(\bar{s} = s' | s_0 = s) = p_{\pi^{sp}}(\bar{s} = s' | s_0 = s)$.

Definition 13 (Mild stochasticity (2))**.** In MDP $\mathcal{M}$, the optimal policy π^* does not visit the same state more than once: For all $s \in \mathcal{S}$ such that $\rho_{\pi^*}(s) > 0$, it holds $\rho_{\pi^*}(s) = 1$, where $\rho_\pi(s) \triangleq \mathbb{E}_{s_0 \sim \rho_0(S), a \sim \pi(A|s), s' \sim (S|s,a)} \left[\sum_{t=1}^{T} \mathbb{I}\left(s_t = s\right) \right]$ is the state-visitation count.

In other words, we assume that the optimal policy does not have a cycle. One common property of MDP that meets this condition is that the reward disappearing after being acquired by the agent. We note that this assumption holds for many practical environments. In fact, in many cases as well as *Atari*, *DeepMind Lab*, etc.

Theorem 7. *For any MDP with the mild stochasticity condition, an optimal policy π^* satisfies the shortest-path constraint: $\pi^* \in \Pi^{SP}$.*

Proof. For simplicity, we prove this based on the option-based view point (see Appendix D.1). By plugging Equation (D.8) and Equation (D.10) into Equation (D.4), we can re-write the Bellman equation of the value function $V^\pi(s)$ as follows:

$$V^\pi(s) = \sum_{o \in \mathcal{O}} Pr[E(o,s)] \left[r_s^o + \sum_{s'} P_{ss'}^o V^\pi(s') \right] \tag{D.21}$$

$$= \sum_{s' \in \mathcal{S}^{\text{IR}}} p_\pi(\bar{s} = s' | s_0 = s) \left[R(s') \mathbb{E}^{\tau \sim \pi}(\gamma^{\ell(\tau)} | s_0 = s, \bar{s} = s') + \gamma P_{s,s'}^\pi V^\pi(s') \right] \tag{D.22}$$

$$= \sum_{s' \in \mathcal{S}^{\text{IR}}} p_\pi(\bar{s} = s' | s_0 = s) \left[R(s') P_{s,s'}^\pi + \gamma P_{s,s'}^\pi V^\pi(s') \right], \tag{D.23}$$

$$= \sum_{s' \in \mathcal{S}^{\text{IR}}} p_\pi(\bar{s} = s' | s_0 = s) P_{s,s'}^\pi \left[R(s') + \gamma V^\pi(s') \right], \tag{D.24}$$

where $\bar{s}$ is the first rewarding state that agent encounters. Intuitively, $p_\pi(\bar{s} = s' | s_0 = s)$ means the probability that the s' is the first rewarding state that policy π encounters when it starts from s. From Equation (D.13), our goal is to show:

$$\pi^* \in \Pi^{\text{SP}} = \{ \pi \mid \forall (s, s') \in \mathcal{T}_{\Phi,\text{nr}}^\pi, P_{s,s'}^\pi = P_{s,s'}^* \}, \tag{D.25}$$

where $P_{s,s'}^* = \max_\pi P_{s,s'}^\pi$.

We will prove Equation (D.25) by contradiction. Suppose π^* is an optimal policy such that $\pi^* \notin \Pi^{\text{SP}}$. Then,

$$\exists(\hat{s}, \hat{s}' \in \mathcal{T}^{\pi^*}_{\Phi,\text{nr}}) \text{ s.t. } P^{\pi^*}_{\hat{s},\hat{s}'} \neq P^*_{\hat{s},\hat{s}'}. \tag{D.26}$$

Recall the definition: $P^*_{s,s'} = \max_\pi P^\pi_{s,s'}$. Then, for any π, the following statement is true.

$$P^\pi_{s,s'} \neq P^*_{s,s'} \leftrightarrow P^\pi_{s,s'} < P^*_{s,s'}. \tag{D.27}$$

Thus, we have

$$P^{\pi^*}_{\hat{s},\hat{s}'} < P^*_{\hat{s},\hat{s}'} \tag{D.28}$$

Let $\pi_{sp} \in \Pi^{\text{SP}}$ be a shortest path policy that preserves stochastic dynamics from Definition 12. Then, we have

$$P^{\pi^*}_{\hat{s},\hat{s}'} < P^*_{\hat{s},\hat{s}'} = P^{\pi_{sp}}_{\hat{s},\hat{s}'}. \tag{D.29}$$

Then, let's compose a new policy $\hat{\pi}$:

$$\hat{\pi}(a|s) = \begin{cases} \pi_{sp}(a|s) & \text{if } \exists\, \tau \in \mathcal{T}^{\pi_{sp}}_{\hat{s},\hat{s}',\text{nr}} \text{ s.t. } s \in \tau \\ \pi^*(a|s) & \text{otherwise} \end{cases}. \tag{D.30}$$

Now consider a path $\tau_{\hat{s}\to\hat{s}'}$ that agent visits $\hat{s}$ at time $t = i$ and transitions to $\hat{s}'$ at time $t = j > i$ while not visiting any rewarding state from $t = i$ to $t = j$ with non-zero probability (*i.e.*, $p_{\pi_{\text{sp}}}(\tau) > 0$). We can define a set of such paths as follows:

$$\hat{\mathcal{T}}_{\hat{s}\to\hat{s}'} = \{\tau \mid \exists(i < j), s_i = \hat{s}, s_j = \hat{s}', \{s_t\}_{i<t<j} \cap \mathcal{S}^{\text{IR}} = \emptyset, p_{\pi_{\text{sp}}}(\tau) > 0\}. \tag{D.31}$$

To reiterate the definitions from Definition 6: $\mathcal{S}^{\text{IR}} = \{s \mid R(s) > 0 \text{ or } \rho(s) > 0\}$ is the union of all initial and rewarding states, and $\Phi^\pi = \{(s, s') \mid s, s' \in \mathcal{S}^{\text{IR}}, \rho(s) > 0, \mathcal{T}^\pi_{s,s',\text{nr}} \neq \emptyset\}$ is the subset of $\mathcal{S}^{\text{IR}}$ such that agent may roll out.

From Definition 13 and Equation (D.30), the likelihood of a path τ under policy $\hat{\pi}$ is given as follows:

$$p_{\hat{\pi}}(\tau) = \begin{cases} p_{\pi^*}(\tau \in \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'}) p_{\pi_{\text{sp}}}(\tau | \tau \in \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'}) & \text{for } \tau \in \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'} \\ p_{\pi^*}(\tau) & \text{otherwise} \end{cases}, \tag{D.32}$$

where $p_{\hat{\pi}}(\tau)$ is the likelihood of trajectory τ under policy $\hat{\pi}$, $p_{\hat{\pi}}(\tau \in \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'}) = \int_{\tau\in\hat{\mathcal{T}}_{\hat{s}\to\hat{s}'}} p_{\hat{\pi}}(\tau) d\tau$ ensures the likelihood $\hat{\pi}(\tau)$ to be a valid probability density function (*i.e.*, $\int p_{\hat{\pi}}(\tau) d\tau = 1$). From the path $\tau_{\hat{s}\to\hat{s}'}$ and i, j, we will choose two states $s_{\text{ir}}, s'_{\text{ir}} \sim \tau_{\hat{s}\to\hat{s}'}$, where

$$s_{\text{ir}} = \max_t(s_t | s_t \in \mathcal{S}^{\text{IR}}, t \leq i), \quad s'_{\text{ir}} = \min_t(s_t | s_t \in \mathcal{S}^{\text{IR}}, j \leq t). \tag{D.33}$$

Note that such s_{ir} and s'_{ir} always exist in $\tau_{\hat{s}\to\hat{s}'}$ since the initial state and the terminal state satisfy the condition to be s_{ir} and s'_{ir}.

Then, we can show that the path between s_{ir} and s'_{ir} is **not** a shortest-path. Recall the definition

of $D^{\pi}_{\text{nr}}(s, s')$ (Definition 3):

$$D^{\pi^*}_{\text{nr}}(s_{\text{ir}}, s'_{\text{ir}}) := \log_\gamma \left(\mathbb{E}_{\tau\sim\pi^*:\ \tau\in\mathcal{T}^{\pi^*}_{s_{\text{ir}},s'_{\text{ir}},\text{nr}}} \left[\gamma^{\ell(\tau)}\right] \right) \tag{D.34}$$

$$= \log_\gamma \left(\mathbb{E}_{\tau\sim\pi^*} \underbrace{\left[\gamma^{\ell(\tau)} \mid \tau \in \mathcal{T}^{\pi^*}_{s_{\text{ir}},s'_{\text{ir}},\text{nr}}\right]}_{\clubsuit} \right) \tag{D.35}$$

where we will use $\clubsuit := \gamma^{\ell(\tau)} \mid \tau \in \mathcal{T}^{\pi^*}_{s_{\text{ir}},s'_{\text{ir}},\text{nr}}$ for a shorthand notation. Then, we have

$$\gamma^{D^{\pi^*}_{\text{nr}}(s_{\text{ir}},s'_{\text{ir}})} := \mathbb{E}_{\tau\sim\pi^*}[\clubsuit] \tag{D.36}$$

$$= p_{\pi^*}(\tau \in \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'})\mathbb{E}_{\tau\sim\pi^*}\left[\clubsuit \mid \tau \in \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'}\right] + p_{\pi^*}(\tau \notin \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'})\mathbb{E}_{\tau\sim\pi^*}\left[\clubsuit \mid \tau \notin \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'}\right] \tag{D.37}$$

$$\text{(From Definition 5)} \quad < p_{\pi^*}(\tau \in \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'})\mathbb{E}_{\tau\sim\pi_{\text{sp}}}\left[\clubsuit \mid \tau \in \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'}\right] + p_{\pi^*}(\tau \notin \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'})\mathbb{E}_{\tau\sim\pi^*}\left[\clubsuit \mid \tau \notin \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'}\right] \tag{D.38}$$

$$\text{(From Equation (D.32))} \quad = p_{\hat{\pi}}(\tau \in \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'})\mathbb{E}_{\tau\sim\hat{\pi}}\left[\clubsuit \mid \tau \in \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'}\right] + p_{\hat{\pi}}(\tau \notin \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'})\mathbb{E}_{\tau\sim\hat{\pi}}\left[\clubsuit \mid \tau \notin \hat{\mathcal{T}}_{\hat{s}\to\hat{s}'}\right] \tag{D.39}$$

$$= \mathbb{E}_{\tau\sim\hat{\pi}}[\clubsuit] = \gamma^{D^{\hat{\pi}}_{\text{nr}}(s_{\text{ir}},s'_{\text{ir}})} \tag{D.40}$$

$$\iff D^{\pi^*}_{\text{nr}}(s_{\text{ir}}, s'_{\text{ir}}) > D^{\hat{\pi}}_{\text{nr}}(s_{\text{ir}}, s'_{\text{ir}}) \tag{D.41}$$

where Ineq. (D.41) is given by the fact that $\gamma < 1$. Then, $P^{\pi^*}_{s_{\text{ir}},s'_{\text{ir}}} < P^{\hat{\pi}}_{s_{\text{ir}},s'_{\text{ir}}}$.

From Equation (D.24), we have

$$V^{\hat{\pi}}(s_{\text{ir}}) = \sum_{s' \in \mathcal{S}^{\text{IR}}} p_{\hat{\pi}}(\bar{s} = s' \mid s_0 = s_{\text{ir}}) P^{\hat{\pi}}_{s_{\text{ir}},s'} \left[R(s') + \gamma V^{\hat{\pi}}(s')\right] \tag{D.42}$$

$$= p_{\hat{\pi}}(\bar{s} = s'_{\text{ir}} \mid s_0 = s_{\text{ir}}) P^{\hat{\pi}}_{s_{\text{ir}},s'_{\text{ir}}} \left[R(s'_{\text{ir}}) + \gamma V^{\hat{\pi}}(s'_{\text{ir}})\right] + \sum_{s' \in \mathcal{S}^{\text{IR}} \setminus s'_{\text{ir}}} p_{\hat{\pi}}(\bar{s} = s' \mid s_0 = s_{\text{ir}}) P^{\hat{\pi}}_{s_{\text{ir}},s'} \left[R(s') + \gamma V^{\hat{\pi}}(s')\right] \tag{D.43}$$

$$= p_{\pi^*}(\bar{s} = s'_{\text{ir}} \mid s_0 = s_{\text{ir}}) P^{\hat{\pi}}_{s_{\text{ir}},s'_{\text{ir}}} \left[R(s'_{\text{ir}}) + \gamma V^{\pi^*}(s'_{\text{ir}})\right] + \sum_{s' \in \mathcal{S}^{\text{IR}} \setminus s'_{\text{ir}}} p_{\pi^*}(\bar{s} = s' \mid s_0 = s_{\text{ir}}) P^{\pi^*}_{s_{\text{ir}},s'} \left[R(s') + \gamma V^{\pi^*}(s')\right] \tag{D.44}$$

$$> p_{\pi^*}(\bar{s} = s'_{\text{ir}} \mid s_0 = s_{\text{ir}}) P^{\pi^*}_{s_{\text{ir}},s'_{\text{ir}}} \left[R(s'_{\text{ir}}) + \gamma V^{\pi^*}(s'_{\text{ir}})\right] + \sum_{s' \in \mathcal{S}^{\text{IR}} \setminus s'_{\text{ir}}} p_{\pi^*}(\bar{s} = s' \mid s_0 = s_{\text{ir}}) P^{\pi^*}_{s_{\text{ir}},s'} \left[R(s') + \gamma V^{\pi^*}(s')\right] \tag{D.45}$$

$$= \sum_{s' \in \mathcal{S}^{\text{IR}}} p_{\pi^*}(\bar{s} = s' \mid s_0 = s_{\text{ir}}) P^{\pi^*}_{s_{\text{ir}},s'} \left[R(s') + \gamma V^{\pi^*}(s')\right] \tag{D.46}$$

$$= V^*(s_{\text{ir}}), \tag{D.47}$$

where Eq. (D.44) holds from the *mild-stochasticity (1)* and *mild-stochasticity (2)* assumption, and Ineq. (D.45) holds because $P^{\hat{\pi}}_{s_{\text{ir}},s'_{\text{ir}}} > P^{\pi^*}_{s_{\text{ir}},s'_{\text{ir}}}$ and $R(s') + \gamma V^{\pi^*}(s') > 0$ from the non-negative optimal value assumption (See Section 5.3). Finally, this is a contradiction since the optimal value function $V^*(s)$ should be the maximum. □

D.4 Experiment details of *MiniGrid* domain

D.4.1 Environment

MiniGrid is a 2D grid-world environment with diverse predefined tasks [Chevalier-Boisvert et al., 2018]. It has several challenging features such as pictorial observation, random initialization of the agent and the goal, complex action space and transition dynamics involving agent's orientation of movement and changing object status via interaction (e.g., key-door).

State Space. An observation $\mathbf{s}_t$ is represented as $H \times W \times C$ tensor, where H and W are the height and width of map respectively, and C is features of the objects in the grid. The (h, w)-th element of observation tensor is $(type,\ color,\ status)$ of the object and for the coordinate of agent, the (h, w)-th element is $(type,\ 0,\ direction)$. The map size (*i.e.*, $H \times W$) varies depending on the task; *e.g.*, for *FourRooms-7×7* task, the map size is 7×7.

Action Space and transition dynamics The episode terminates in 100 steps, and the episode may terminate earlier if the agent reaches the goal before 100 steps. The action space consists of seven discrete actions with the following transitions.

- `Turn-Counter-Clockwise`: change the $direction$ counter-clockwise by 90 degree.
- `Turn-Clockwise`: change the $direction$ clockwise by 90 degree.
- `Move-Forward`: move toward $direction$ by 1 step unless blocked by other objects.
- `Pick-up-key`: pickup the key if the key is in front of the agent.
- `Drop-the-key`: drop the key in front of the agent.
- `Open/Close-doors`: open/close the door if the door is in front of the agent.
- `Optional-action`: not used

Reward function. The reward is given only if the agent reaches the goal location, and the reward magnitude is $1 - 0.9(\text{length of episode}/\text{maximum step for episode})$. Thus, the agent can maximize the reward by reaching to the goal location in shortest time.

D.4.2 Tasks

In *FourRooms-7×7* and *FourRooms-11×11*, the map structure has four large rooms, and the agent needs to reach to the goal. In *KeyDoors-7×7* and *KeyDoors-11×11*, the agent needs to pick up the key, go to the door, and open the door before reaching to the goal location.

D.4.3 Architecture and hyper-parameters

We used a simple CNN architecture similar to [Mnih et al., 2015] for policy network. The network consists of `Conv1(16x2x2-1/SAME)-CReLU-Conv2(8x2x2-1/SAME)-CReLU-Conv3(8x2x2-1/SAME)-CReLU-FC(512)-FC(action-dimension)`, where `SAME` padding ensures the input and output have the same size (*i.e.*, width and height) and CReLU [Shang et al., 2016] is a non-linear activation function applied after each layer. We used Adam [Kingma and Ba, 2014] optimizer to optimize the policy network.

For hyper-parameter search, we swept over a set of hyper-parameters specified in Table D.1, and chose the best one in terms of the mean AUC over all the tasks, which is also summarized in Table D.1.

D.5 Experiment details of *DeepMind Lab* domain

D.5.1 Environment

DeepMind Lab is a 3D-game environment with first-person view. Along with random initialization of the agent and the goal, complex action space including directional change, random change of texture, color and maze structure are features that make tasks in *DeepMind Lab* hard to be learned.

State Space. A state $\mathbf{s}_t$ has the dimension of $84 \times 84 \times 3$. The state is given as a first-person view of the map structure. We resized the RGB image into $84 \times 84 \times 3$ RGB image, and normalized by dividing the pixel value by 255.

Action Space and transition dynamics The episode terminates after the fixed number of steps regardless of goal being achieved. The original action space consists of seven discrete actions: `Move-Forward`, `Move-Backward`, `Strafe Left`, `Strafe Right`, `Look Left`, `Look Right`, `Look Left and Move-Forward`, `Look Right and Move-Forward`. In our experiment, we used eight discrete actions with the additional action `Fire` as in [Higgins et al., 2017, Vezhnevets et al., 2017, Savinov et al., 2018b, Espeholt et al., 2018, Khetarpal and Precup, 2018].

D.5.2 Tasks

We tested our agent and compared methods on three standard tasks in *DeepMind Lab*: *GoalSmall*, *GoalLarge*, and *ObjectMany* which correspond to `explore_goal_locations_small`, `explore_goal_locations_large`, and `explore_object_rewards_many`, respectively. *GoalSmall* and *GoalLarge* has a single goal in the maze, but the size of the maze is larger in *GoalLarge* than *GoalSmall*. The agent and goal locations are randomly set in the beginning of the episode and the episode length is fixed to 1,350 steps for *GoalSmall* and 1,800 steps for *GoalLarge*. When the agent reaches the goal, it positively rewards the agent and the agent is re-spawned in a random location without terminating the episode, such that the agent can reach to the goal multiple times within a single episode. Thus, the agent's goal is to reach to the goal location as many times as possible within the episode length. *ObjectMany* has multiple objects in the maze, where reaching to the object positively rewards the agent and the object disappears. The episode length is fixed to 1,800 steps. The agent's goal is to gather as many object as possible within the episode length.

Require: Hyperparameters: $k \in \mathbb{N}$, Positive bias $\Delta^+ \in \mathbb{N}$, Negative bias $\Delta^- \in \mathbb{N}$
Initialize $t_{\text{anc}} \leftarrow 0$.
Initialize $S_{\text{anc}} = \emptyset$, $S_+ = \emptyset$, $S_- = \emptyset$.
while $t_{\text{anc}} < T$ **do**
    $S_{\text{anc}} = S_{\text{anc}} \cup \{s_{t_{\text{anc}}}\}$.
    $t_+ = \text{Uniform}(t_{\text{anc}} + 1, t_{\text{anc}} + k)$.
    $t_- = \text{Uniform}(t_{\text{anc}} + k + \Delta^-, T)$.
    $S_+ = S_+ \cup \{s_{t_+}\}$.
    $S_- = S_- \cup \{s_{t_-}\}$.
    $t_{\text{anc}} = \text{Uniform}(t_+ + 1, t_+ + \Delta^+)$.
end while
Return S_{anc}, S_+, S_-

Program D.1: Sampling the triplet data from an episode for RNet training

D.5.3 Reachability network Training

Similar to Savinov et al. [2018b], we used the following contrastive loss for training the reachability network:

$$\mathcal{L}_{\text{Rnet}} = -\log\left(\text{Rnet}_{k-1}(s_{\text{anc}}, s_+)\right) - \log\left(1 - \text{Rnet}_{k-1}(s_{\text{anc}}, s_-)\right), \tag{D.48}$$

where s_{anc}, s_+, s_- are the anchor, positive, and negative samples, respectively. The anchor, positive and negative samples are sampled from the same episode, and their time steps are sampled according to **Algorithm** D.1. The RNet is trained in an off-policy manner from the replay buffer with the size of 60K environment steps collecting agent's online experience. We found that adaptive scheduling of RNet is helpful for faster convergence of RNet. Out of 20M total environment steps, for the first 1M, 1M, and 18M environment steps, we updated RNet every $6K$, $12K$, and $36K$ environment steps, respectively. For all three environments of *DeepMind Lab*, RNet accuracy was ~ 0.9 after 1M steps.

Multiple tolerance. In order to improve the stability of Reachability prediction, we used the statistics over multiple samples rather than using a single-sample estimate as suggested in Equation (6.12). As a choice of sampling method, we simply used multiples of tolerance. In other words, given $s_{t-(k+\Delta t)}$ and s_t as inputs for reachability network, we instead used $s_{t-(k+n\Delta t)}$ and s_t where $1 \leq n \leq N_{\Delta t}$, $n \in \mathbb{N}$ and $N_{\Delta t}$ is the number of tolerance samples. We used 90-percentile of $N_{\Delta t}$ outputs of reachability network, $\text{Rnet}_{k-1}(s_{t-(k+n\Delta t)}, s_t)$, as in [Savinov et al., 2018b] to get the representative of the samples.

D.5.4 Architecture and hyper-parameters

Following [Savinov et al., 2018b], we used the same CNN architecture used in [Mnih et al., 2015].

For **SPRL**, we used a smaller reachability network (RNet) architecture compared to **ECO** to reduce the training time. The RNet is based on siamese architecture with two branches. Following [Savinov et al., 2018b], **ECO** used Resnet-18 [He et al., 2016] architecture with 2-2-2-2 residual blocks and 512-dimensional output fully-connected layer to implement each branch. For **SPRL**, we used Resnet-12 with 2-2-1 residual blocks and 512-dimensional output fully-connected layer to implement each branch. The RNet takes two states as inputs, and each state is fed into each branch. The outputs of the two branches are concatenated and forwarded to three[1] 512-dimensional fully-connected layers to produce one-dimensional sigmoid output, which predicts the reachability between two state inputs. We also resized the observation to the same dimension as policy (*i.e.*, $84 \times 84 \times 3$, which is smaller than the original $120 \times 160 \times 3$ used in [Savinov et al., 2018b]).

For all the baselines (*i.e.*, **PPO**, **ECO**, **ICM**, and **GT-Grid**), we used the best hyperparameter used in [Savinov et al., 2018b]. For **SPRL**, we searched over a set of hyperparameters specified in Table D.2, and chose the best one in terms of the mean AUC over all the tasks, which is also summarized in Table D.2.

D.6 Experiment details of *Atari* domain

D.6.1 Environment

Atari is an important and prominent benchmark in deep reinforcement learning with a high-dimensional visual input. One of the main benefit of using *Atari* as a testbed is that it covers not only navigational tasks but various tasks such as avoiding and destroying enemies by firing a bullet or changing the map structure using bombs as explained in [Bellemare et al., 2013]. Because of the diversity of the task *Atari* is covering, solving *Atari* shows that the algorithm has a certain degree of generality.

There exists a variety of preprocessing details for *Atari*. We mostly followed the implementation of OpenAI Baselines [Dhariwal et al., 2017]. For detailed information, see Table D.3.

State Space. A state $\mathbf{s}_t$ is represented as $84 \times 84 \times 4$. We stacked 4 consecutive frames achieved by taking the same action 4 times in a row and resized RGB image into $84 \times 84 \times 1$ gray image, and normalized by dividing the pixel value by 255.

[1] Savinov et al. [2018b] used four 512-dimensional fully-connected layers.

Action Space and transition dynamics The episode terminates when the agent loses all of the lives given. The action space consists of eighteen discrete actions as in [Bellemare et al., 2013].

D.6.2 Various Tasks of *Atari*: Navigational and Non-navigational

We tested our agent and compared methods on navigational and **non-navigational tasks** in *Atari*: *Montezuma's Revenge*, *Freeway*, *Ms.Pacman*, *Gravitar*, *Seaquest*, *HERO*. *Montezuma's Revenge* is a famous game as a hard exploration game in *Atari*. An agent should pick up the items such as key to open the door or knife to destroy the enemy. In *Freeway*, an agent should cross the road while avoiding the car. *Ms.Pacman* is a game where an agent should eat the items and avoid enemy. Three games mentioned until now have a navigational feature meaning that the agent can move toward a certain coordinate to get the score. However, three games to be mentioned have a non-navigational feature meaning that agent should not only get to a certain coordinate but also use specific action to get the score. In *Gravitar* and *Seaquest*, an agent should shoot a bullet. In *HERO*, an agent can install a bomb to break the wall and move forward. By adding *Gravitar*, *Seaquest*, and *HERO* in our testbed, we evaluated how **SPRL** performs in non-navigational tasks.

D.6.3 Reachability network Training

We followed the details of reachability network training for *DeepMind Lab* except for 1) replay buffer size and 2) reachability network training frequency. We changed the size of the replay buffer for reachability network from $60K$ environment steps to $30K$ environment steps. To avoid overfitting of the reachability network, we enlarged reachability network training frequency from 6K environment steps to 150K environment steps after the initial 1M environment steps.

Stabilizing Reachability Network. In some of the games of *Atari*, within a task exists a distributional shift in the state space that hinders stable reachability network training. Therefore we had to use some techniques to mitigate instability problem of reachability network: Weight decay and Label smoothing. We used weight decay with the factor of 0.03 and label smoothing of 0.1.

D.6.4 Architecture and hyper-parameters

For the policy architecture, we used the same CNN architecture used in Mnih et al. [2015]. For reachability network, we used the same architecture used for *DeepMind Lab* except for the input layer. For the input layer, we used $(\mathbf{s}_t, \mathbf{s}_t - \mathbf{s}_{t-k})$ instead of $(\mathbf{s}_t, \mathbf{s}_{t-k})$. This change helped the reachability network to avoid suffering from overfitting when the distribution shift in the state space occurs.

For hyper-parameter search, we swept over a set of hyper-parameters specified in Table D.4, and chose the best one in terms of the mean AUC over all the tasks, which is also summarized in Table D.4.

PPO		
Hyperparameters	**Sweep range**	**Final value**
Learning rate	0.001, 0.002, 0.003	0.003
Entropy	0.003, 0.005, 0.01, 0.02, 0.05	0.01
ICM		
Hyperparameters	**Sweep range**	**Final value**
Learning rate	0.001, 0.002, 0.003	0.003
Entropy	-	0.01
Forward/Inverse model loss weight ratio	0.2, 0.5, 0.8, 1.0	0.8
Curiosity module loss weight	0.03, 0.1, 0.3, 1.0	0.3
ICM bonus weight	0.1, 0.3, 1.0, 3.0	0.1
GT-Grid		
Hyperparameters	**Sweep range**	**Final value**
Learning rate	0.001, 0.002, 0.003	0.003
Entropy	-	0.01
GT-Grid bonus weight	0.003, 0.01, 0.03, 0.1, 0.3	0.01
ECO		
Hyperparameters	**Sweep range**	**Final value**
Learning rate	-	0.003
Entropy	-	0.01
k	3, 5, 7	3
ECO bonus weight	0.001, 0.002, 0.005, 0.01	0.001
SPRL		
Hyperparameters	**Sweep range**	**Final value**
Learning rate	0.003, 0.01	0.01
Entropy	-	0.01
k	2, 5	2
Tolerance (Δt)	-	1
Negative bias (Δ^-)	10, 20	20
Positive bias (Δ^+)	-	5
Cost scale (λ)	0.001, 0.002, 0.005	0.002
$N_{\Delta t}$	30, 60	60

Table D.1: The range of hyperparameters sweeped over and the final hyperparameters used in *MiniGrid* domain.

Hyperparameters for SPRL	**Sweep range**	**Final value**
Learning rate	-	0.0003
Entropy	-	0.004
k	3, 10, 30	10
Tolerance (Δt)	1, 3, 5	1
Negative bias (Δ^-)	5, 10, 20	20
Positive bias (Δ^+)	-	5
Cost scale (λ)	0.02, 0.06, 0.2	0.06
Optimizer	-	Adam
$N_{\Delta t}$	-	200

Table D.2: The range of hyperparameters sweeped over and the final hyperparameters used for our **SPRL** method in *DeepMind Lab* domain.

Parameter	**Value**
Image Width	84
Image Height	84
Grayscaling	Yes
Number of Actions	18
Action Repetitions	4
Frame Stacking	4
End of episode when life lost	No
Reward Clipping	[-1,1]
Discount(γ)	0.99
Max Episode Length	10000
Number of parallel workers	12

Table D.3: Preprocessing details for *Atari*

PPO		
Hyperparameters	**Sweep range**	**Final value**
Learning rate	0.0001, 0.0002, 0.0003, 0.0005	0.0005
Entropy	0.001, 0.003, 0.005, 0.01, 0.03	0.01
ICM		
Hyperparameters	**Sweep range**	**Final value**
Learning rate	0.0005	0.0005
Entropy	-	0.01
Forward/Inverse model loss weight ratio	1.0	1.0
Curiosity module loss weight	1.0	1.0
ICM bonus weight	0.0001, 0.0003, 0.001, 0.003, 0.01	0.01
ECO		
Hyperparameters	**Sweep range**	**Final value**
Learning rate	0.0001, 0.0003, 0.0005	0.0005
Entropy	-	0.01
ECO bonus weight	0.001, 0.003, 0.01, 0.03, 0.1	0.001
SPRL		
Hyperparameters	**Sweep range**	**Final value**
Learning rate	0.0003, 0.0005	0.0005
Entropy	-	0.01
SPRL cost scale (λ)	0.01, 0.03, 0.05, 0.1	0.05
Reachability network (for **ECO** and **SPRL**)		
k	5, 8, 12, 15	12
Tolerance (Δt)	-	1
Negative bias (Δ^-)	80, 100, 120	80
Positive bias (Δ^+)	-	5
$N_{\Delta t}$	30, 50, 100, 200, 400	200

Table D.4: The range of hyperparameters sweeped over and the final hyperparameters used in *Atari* domain.

BIBLIOGRAPHY

David Abel, David Hershkowitz, and Michael Littman. Near optimal behavior via approximate state abstraction. In *International Conference on Machine Learning*, pages 2915–2923, 2016.

David Abel, Dilip Arumugam, Lucas Lehnert, and Michael Littman. State abstractions for lifelong reinforcement learning. In *International Conference on Machine Learning*, pages 10–19, 2018.

David Andre and Stuart J. Russell. Programmable reinforcement learning agents. In *NIPS*, 2000.

David Andre and Stuart J. Russell. State abstraction for programmable reinforcement learning agents. In *AAAI/IAAI*, 2002.

Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *ICML*, 2017.

Takao Asano, Tetsuo Asano, Leonidas Guibas, John Hershberger, and Hiroshi Imai. Visibility-polygon search and euclidean shortest paths. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*, pages 155–164. IEEE, 1985.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

Dimitri P Bertsekas and John N Tsitsiklis. An analysis of stochastic shortest path problems. *Mathematics of Operations Research*, 16(3):580–595, 1991.

Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Proceedings of 1995 34th IEEE Conference on Decision and Control*, volume 1, pages 560–564. IEEE, 1995.

Dimitri P Bertsekas, David A Castanon, et al. Adaptive aggregation methods for infinite horizon dynamic programming. 1988.

M.K. Bloch. Hierarchical reinforcement learning in the taxicab domain. *(Report No. CCA-TR-2009-02). Ann Arbor, MI: Center for Cognitive Architecture, University of Michigan*, 2009.

Craig Boutilier, Richard Dearden, Moises Goldszmidt, et al. Exploiting structure in policy construction. In *IJCAI*, volume 14, pages 1104–1113, 1995.

Leo Breiman. *Classification and regression trees*. Routledge, 1984.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*, 2018.

John Canny. A voronoi method for the piano-movers problem. In *Robotics and Automation. Proceedings. 1985 IEEE International Conference on*, volume 2, pages 530–535. IEEE, 1985.

John Canny. A new algebraic method for robot motion planning and real geometry. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 39–48. IEEE, 1987.

Wilka Carvalho, Anthony Liang, Kimin Lee, Sungryull Sohn, Honglak Lee, Richard L Lewis, and Satinder Singh. Reinforcement learning for sparse-reward object-interaction tasks in first-person simulated 3d environments. *arXiv preprint arXiv:2010.15195*, 2020.

Luis Castillo, Juan Fdez-Olivares, Óscar García-Pérez, and Francisco Palao. Temporal enhancements of an htn planner. In *Conference of the Spanish Association for Artificial Intelligence*, pages 429–438. Springer, 2005.

Pablo Samuel Castro. Scalable methods for computing state similarity in deterministic markov decision processes. *arXiv preprint arXiv:1911.09291*, 2019.

Devendra Singh Chaplot, Kanthashree Mysore Sathyendra, Rama Kumar Pasumarthi, Dheeraj Rajagopal, and Ruslan Salakhutdinov. Gated-attention architectures for task-oriented language grounding. In *AAAI*, 2018.

Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and Andrew Rabinovich. Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. In *International Conference on Machine Learning*, pages 794–803. PMLR, 2018.

Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for openai gym. https://github.com/maximecb/gym-minigrid, 2018.

Jongwook Choi, Yijie Guo, Marcin Moczulski, Junhyuk Oh, Neal Wu, Mohammad Norouzi, and Honglak Lee. Contingency-aware exploration in reinforcement learning. *arXiv preprint arXiv:1811.01483*, 2018.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

Bruno Da Silva, George Konidaris, and Andrew Barto. Learning parameterized skills. *arXiv:1206.6398*, 2012.

Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational intelligence*, 5(2):142–150, 1989.

Thomas L Dean, Robert Givan, and Sonia Leach. Model reduction techniques for computing approximately optimal solutions for markov decision processes. *arXiv preprint arXiv:1302.1533*, 2013.

Misha Denil, Sergio Gómez Colmenarejo, Serkan Cabi, David Saxton, and Nando de Freitas. Programmable agents. *arXiv preprint arXiv:1706.06383*, 2017.

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.

Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *J. Artif. Intell. Res.(JAIR)*, 13:227–303, 2000.

Carlos Diuk, Andre Cohen, and Michael L Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pages 240–247, 2008.

Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. Neural logic machines. In *ICLR*, 2019.

Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl ^{2}: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

Kutluhan Erol. *Hierarchical task network planning: formalization, analysis, and implementation*. PhD thesis, 1996.

Kutluhan Erol, James A Hendler, and Dana S Nau. Umcp: A sound and complete procedure for hierarchical task-network planning. In *AIPS*, volume 94, pages 249–254, 1994.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

Richard Evans and Edward Grefenstette. Learning Explanatory Rules from Noisy Data. *arXiv preprint arXiv:1711.04574*, 2017.

Bernard Faverjon and Pierre Tournassoud. A local based approach for path planning of manipulators with a high number of degrees of freedom. In *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*, volume 4, pages 1152–1159. IEEE, 1987.

Norm Ferns, Prakash Panangaden, and Doina Precup. Metrics for finite markov decision processes. In *UAI*, volume 4, pages 162–169, 2004.

Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017.

Chelsea Finn, Kelvin Xu, and Sergey Levine. Probabilistic model-agnostic meta-learning. In *NeurIPS*, pages 9516–9527, 2018.

Evelyn Fix. *Discriminatory analysis: nonparametric discrimination, consistency properties*, volume 1. USAF school of Aviation Medicine, 1985.

Lester R Ford Jr. Network flow theory. Technical report, Rand Corp Santa Monica Ca, 1956.

Jeremy Frank and Ari Jónsson. Constraint-based attribute and interval planning. *Constraints*, 8(4): 339–364, 2003.

Mohammad Ghavamzadeh and Sridhar Mahadevan. Hierarchical policy gradient algorithms. In *ICML*, pages 226–233, 2003.

Behzad Ghazanfari and Matthew E Taylor. Autonomous extracting a hierarchical structure of tasks in reinforcement learning and multi-task reinforcement learning. *arXiv preprint arXiv:1709.04579*, 2017.

Robert Givan, Thomas Dean, and Matthew Greig. Equivalence notions and model minimization in markov decision processes. *Artificial Intelligence*, 147(1-2):163–223, 2003.

Xiaoxiao Guo, Satinder Singh, Richard Lewis, and Honglak Lee. Deep learning for reward design to improve monte carlo tree search in atari games. *arXiv preprint arXiv:1604.07095*, 2016.

Abhishek Gupta, Russell Mendonca, YuXuan Liu, Pieter Abbeel, and Sergey Levine. Meta-reinforcement learning of structured exploration strategies. *arXiv preprint arXiv:1802.07245*, 2018.

Abhishek Gupta, Vikash Kumar, Corey Lynch, Sergey Levine, and Karol Hausman. Relay policy learning: Solving long-horizon tasks via imitation and reinforcement learning. *arXiv preprint arXiv:1910.11956*, 2019.

Izzeddin Gur, Ulrich Rueckert, Aleksandra Faust, and Dilek Hakkani-Tur. Learning to navigate the web. *arXiv preprint arXiv:1812.09195*, 2018.

Karol Hausman, Jost Tobias Springenberg, Ziyu Wang, Nicolas Heess, and Martin Riedmiller. Learning an embedding space for transferable robot skills. In *International Conference on Learning Representations*, 2018.

Bradley Hayes and Brian Scassellati. Autonomously constructing hierarchical task networks for planning and human-robot collaboration. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5469–5476. IEEE, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Irina Higgins, Arka Pal, Andrei Rusu, Loic Matthey, Christopher Burgess, Alexander Pritzel, Matthew Botvinick, Charles Blundell, and Alexander Lerchner. Darla: Improving zero-shot transfer in reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1480–1490. JMLR. org, 2017.

Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

Zhiting Hu, Xuezhe Ma, Zhengzhong Liu, Eduard Hovy, and Eric Xing. Harnessing deep neural networks with logic rules. *arXiv preprint arXiv:1603.06318*, 2016.

De-An Huang, Suraj Nair, Danfei Xu, Yuke Zhu, Animesh Garg, Li Fei-Fei, Silvio Savarese, and Juan Carlos Niebles. Neural task graphs: Generalizing to unseen tasks from a single video demonstration. *arXiv preprint arXiv:1807.03480*, 2018.

Zhiao Huang, Fangchen Liu, and Hao Su. Mapping state space using landmarks for universal goal reaching. In *Advances in Neural Information Processing Systems*, pages 1940–1950, 2019.

Sheng Jia, Jamie Ryan Kiros, and Jimmy Ba. DOM-q-NET: Grounded RL on structured language. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HJgd1nAqFX.

Anders Jonsson and Andrew Barto. Causal graph based decomposition of factored mdps. *Journal of Machine Learning Research*, 7(Nov):2259–2301, 2006.

Sham Machandranath Kakade et al. *On the sample complexity of reinforcement learning*. PhD thesis, University of London London, England, 2003.

J Mark Keil and Jorg-R Sack. Minimum decompositions of polygonal objects. In *Machine Intelligence and Pattern Recognition*, volume 2, pages 197–216. Elsevier, 1985.

Khimya Khetarpal and Doina Precup. Attend before you act: Leveraging human visual attention for continual learning. *arXiv preprint arXiv:1807.09664*, 2018.

Taesup Kim, Jaesik Yoon, Ousmane Dia, Sungwoong Kim, Yoshua Bengio, and Sungjin Ahn. Bayesian model-agnostic meta-learning. *arXiv preprint arXiv:1806.03836*, 2018.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Thomas Kipf, Yujia Li, Hanjun Dai, Vinicius Zambaldi, Alvaro Sanchez-Gonzalez, Edward Grefenstette, Pushmeet Kohli, and Peter Battaglia. Compile: Compositional imitation learning and execution. In *International Conference on Machine Learning*, pages 3418–3428. PMLR, 2019.

George Konidaris and Andrew Barto. Autonomous shaping: Knowledge transfer in reinforcement learning. In *Proceedings of the 23rd international conference on Machine learning*, pages 489–496, 2006.

George Konidaris and Andrew G. Barto. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, 2007.

Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

Michael Laskin, Scott Emmons, Ajay Jain, Thanard Kurutach, Pieter Abbeel, and Deepak Pathak. Sparse graphical memory for robust planning. *arXiv preprint arXiv:2003.06417*, 2020.

Alessandro Lazaric. Transfer in reinforcement learning: a framework and a survey. In *Reinforcement Learning*, pages 143–173. Springer, 2012.

Alessandro Lazaric, Marcello Restelli, and Andrea Bonarini. Transfer of samples in batch reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pages 544–551, 2008.

Hoang Le, Nan Jiang, Alekh Agarwal, Miroslav Dudík, Yisong Yue, and Hal Daumé. Hierarchical imitation and reinforcement learning. In *International conference on machine learning*, pages 2917–2926. PMLR, 2018.

Lihong Li, Thomas J Walsh, and Michael L Littman. Towards a unified theory of state abstraction for mdps. In *ISAIM*, 2006.

Xingyu Lin, Harjatin Singh Baweja, George Kantor, and David Held. Adaptive auxiliary task weighting for reinforcement learning. *Advances in neural information processing systems*, 32, 2019.

Changsong Liu, Shaohua Yang, Sari Iaba-Sadiya, Nishant Shukla, Yunzhong He, Song-chun Zhu, and Joyce Chai. Jointly learning grounded task structures from language instruction and visual demonstration. In *EMNLP*, 2016.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. *arXiv preprint arXiv:1802.08802*, 2018.

Joao Loula, Marco Baroni, and Brenden M Lake. Rearranging the familiar: Testing compositional generalization in recurrent networks. *arXiv preprint arXiv:1807.07545*, 2018.

Ajay Mandlekar, Danfei Xu, Roberto Martín-Martín, Silvio Savarese, and Li Fei-Fei. Learning to generalize across long-horizon tasks from human demonstrations. *arXiv preprint arXiv:2003.06085*, 2020.

Neville Mehta, Prasad Tadepalli, and Alan Fern. Autonomous learning of action models for planning. *Advances in Neural Information Processing Systems*, 24:2465–2473, 2011.

Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. In *ICLR*, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Stephen Muggleton. Inductive logic programming. *New Gen. Comput.*, 8(4):295–318, February 1991. ISSN 0288-3635. doi: 10.1007/BF03037089. URL http://dx.doi.org/10.1007/BF03037089.

Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994.

Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 3303–3313, 2018.

Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 968–973. Morgan Kaufmann Publishers Inc., 1999.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.

Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

P Russel Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.

Junhyuk Oh, Satinder Singh, Honglak Lee, and Pushmeet Kohli. Zero-shot task generalization with multi-task deep reinforcement learning. In *ICML*, 2017.

Junhyuk Oh, Yijie Guo, Satinder Singh, and Honglak Lee. Self-imitation learning. In *International Conference on Machine Learning*, pages 3878–3887. PMLR, 2018.

Pierre-Yves Oudeyer and Frederic Kaplan. What is intrinsic motivation? a typology of computational approaches. *Frontiers in neurorobotics*, 1:6, 2009.

Mark Palatucci, D. Pomerleau, Geoffrey E. Hinton, and Tom Michael Mitchell. Zero-shot learning with semantic output codes. In *NIPS*, 2009.

Emilio Parisotto, Jimmy Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *CoRR*, abs/1511.06342, 2015.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

Ronald Parr and Stuart J. Russell. Reinforcement learning with hierarchies of machines. In *NIPS*, 1997.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2778–2787. JMLR. org, 2017.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

Lerrel Pinto and Abhinav Gupta. Learning to push by grasping: Using multiple tasks for effective learning. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 2161–2168. IEEE, 2017.

Doina Precup. Temporal abstraction in reinforcement learning. 2000.

Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009.

Andrei A Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. *arXiv preprint arXiv:1511.06295*, 2015.

Earl D Sacerdoti. The nonlinear nature of plans. Technical report, Stanford Research Institute, Menlo Park, CA, 1975.

Nikolay Savinov, Alexey Dosovitskiy, and Vladlen Koltun. Semi-parametric topological memory for navigation. *arXiv preprint arXiv:1803.00653*, 2018a.

Nikolay Savinov, Anton Raichuk, Raphaël Marinier, Damien Vincent, Marc Pollefeys, Timothy Lillicrap, and Sylvain Gelly. Episodic curiosity through reachability. *ICLR*, 2018b.

Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *International conference on machine learning*, pages 1312–1320, 2015.

Jürgen Schmidhuber. Adaptive confidence and adaptive curiosity. In *Institut fur Informatik, Technische Universitat Munchen, Arcisstr. 21, 800 Munchen 2*. Citeseer, 1991.

J Schulman, F Wolski, P Dhariwal, A Radford, and O Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *ICLR*, 2016.

Dale Schuurmans and Relu Patrascu. Direct value-approximation for factored MDPs. In *NIPS*, pages 1579–1586, 2002.

Wenling Shang, Kihyuk Sohn, Diogo Almeida, and Honglak Lee. Understanding and improving convolutional neural networks via concatenated rectified linear units. In *international conference on machine learning*, pages 2217–2225, 2016.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of bits: An open-domain platform for web-based agents. In *International Conference on Machine Learning*, pages 3135–3144, 2017.

Kyriacos Shiarlis, Markus Wulfmeier, Sasha Salter, Shimon Whiteson, and Ingmar Posner. Taco: Learning task decomposition via temporal alignment for control. In *International Conference on Machine Learning*, pages 4654–4663. PMLR, 2018.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

Sungryull Sohn, Junhyuk Oh, and Honglak Lee. Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies. In *NeurIPS*, pages 7156–7166, 2018.

Sungryull Sohn, Hyunjae Woo, Jongwook Choi, and Honglak Lee. Meta reinforcement learning with autonomous inference of subtask dependencies. In *International Conference on Learning Representations*, 2019.

Sungryull Sohn, Hyunjae Woo, Jongwook Choi, and Honglak Lee. Meta reinforcement learning with autonomous inference of subtask dependencies. *arXiv preprint arXiv:2001.00248*, 2020.

Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on Abstraction, Reformulation, and Approximation*, pages 212–223. Springer, 2002.

Alejandro Suárez-Hernández, Javier Segovia-Aguas, Carme Torras, and Guillem Alenyà. Strips action discovery. *arXiv preprint arXiv:2001.11457*, 2020.

Sainbayar Sukhbaatar, Arthur Szlam, Gabriel Synnaeve, Soumith Chintala, and Rob Fergus. Mazebase: A sandbox for learning from games. *arXiv preprint arXiv:1511.07401*, 2015.

Richard S Sutton. Between mdps and semi-mdps: Learning, planning, and representing knowledge at multiple temporal scales. 1998.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, 1999a.

Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999b.

Austin Tate. Generating project networks. In *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 2*, pages 888–893. Morgan Kaufmann Publishers Inc., 1977.

Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(7), 2009.

Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3): 58–68, 1995.

Chen Tessler, Daniel J Mankowitz, and Shie Mannor. Reward constrained policy optimization. *ICLR*, 2019.

Harm Van Seijen, Mehdi Fatemi, Joshua Romoff, Romain Laroche, Tavian Barnes, and Jeffrey Tsang. Hybrid reward architecture for reinforcement learning. In *NIPS*, pages 5392–5402, 2017.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3540–3549. JMLR. org, 2017.

Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft II: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

Tom Vodopivec, Spyridon Samothrakis, and Branko Ster. On monte carlo tree search and reinforcement learning. *Journal of Artificial Intelligence Research*, 60:881–936, 2017.

Thomas J Walsh and Michael L Littman. Efficient learning of action schemas and web-service descriptions. In *AAAI*, volume 8, pages 714–719, 2008.

Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Aaron Wilson, Alan Fern, Soumya Ray, and Prasad Tadepalli. Multi-task reinforcement learning: a hierarchical bayesian approach. In *Proceedings of the 24th international conference on Machine learning*, pages 1015–1022, 2007.

Danfei Xu, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, and Silvio Savarese. Neural task programming: Learning to generalize across hierarchical tasks. *arXiv preprint arXiv:1710.01813*, 2017.

Haonan Yu, Haichao Zhang, and Wei Xu. A deep compositional framework for human-like language acquisition in virtual environment. *arXiv preprint arXiv:1703.09831*, 2017.

Amy Zhang, Adam Lerer, Sainbayar Sukhbaatar, Rob Fergus, and Arthur Szlam. Composable planning with attributes. *ICML*, 2018.

Yu Zhang and Dit-Yan Yeung. A regularization approach to learning task relationships in multitask learning. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(3):1–31, 2014.

Hankz Hankui Zhuo, Qiang Yang, Derek Hao Hu, and Lei Li. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence*, 174(18):1540–1569, 2010.