Multitask Reinforcement Learning for Zero-shot Generalization with Subtask Dependencies

Sungryull Sohn[†] **Junhyuk Oh**[†] **Honglak Lee**[*,†]
[†]University of Michigan
[*]Google Brain
{srsohn,junhyuk}@umich.edu, honglak@google.com

Abstract

We introduce a new RL problem where the agent is required to execute a given subtask graph which describes a set of subtasks and their dependency. Unlike existing multitask RL approaches that explicitly describe what the agent should do, a subtask graph in our problem only describes properties of subtasks and relationships among them, which requires the agent to perform complex reasoning to find the optimal sequence of subtasks to execute. To tackle this problem, we propose a neural subtask graph solver (NSS) which encodes the subtask graph using a recursive neural network. To overcome the difficulty of training, we propose a novel non-parametric gradient-based policy to pre-train our NSS agent. 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 (1) our method is much more efficient than MCTS and (2) combining MCTS with NSS dramatically improves the search performance.

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 [1, 2] or formal language [3, 4]. However, most of the prior works have focused on task descriptions which explicitly specify what the agent should do, which may not be readily available in real-world applications.

Suppose that we 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 difficult for human users to manually find 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 depedencies of subtasks.

The goal of this paper 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 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 [1]). This allows us to define many complex tasks in a principled way and train the agent to find the

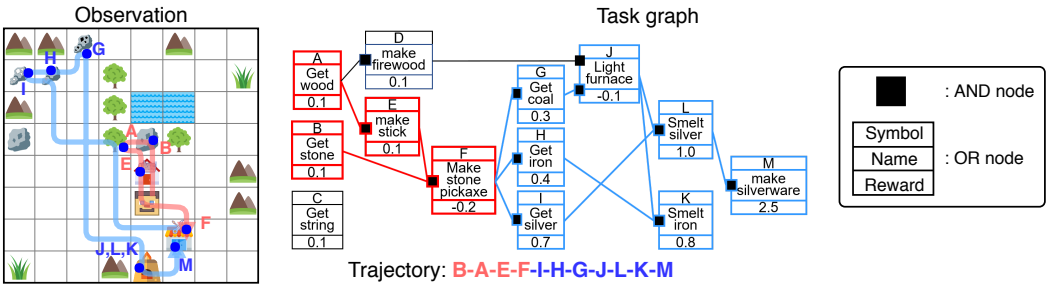

Figure 1: Example task and our agent's trajectory. The agent is required to execute subtasks in the optimal order to maximize the reward within a time limit. The subtask graph describes subtasks with the corresponding rewards (e.g. subtask L gives 1.0 reward) and dependencies between subtasks through AND and OR nodes. For instance, the agent should first get the fire wood (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.

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* (NSS), which encodes a subtask graph using a recursive-reverse-recursive neural network (R3NN) [5] 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 NSS to approximate our novel non-parametric policy called *reward-propagation policy*. The key idea of 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 NSS 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 NSS 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 (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 turns out to significantly improve the performance of MCTS with a much less amount of simulations.

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) [6–8]. 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 [4]. Other approaches used a program as a form of task description rather than a partial policy in the context of multitask RL [1, 3]. 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 *task graph* (see Figure 1), which describes properties of subtasks and dependencies between them. Thus, 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 [9–13]. Our work is built 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.

Classical Search-Based Planning One of the most closely related problem is the planning problem considered in hierarchical task network (HTN) approaches [14–18] 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 [19–23]. 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 Problem Definition

3.1 Preliminary: Multitask Reinforcement Learning and Zero-Shot Generalization

We consider an agent presented with a task drawn from some distribution as in Andreas et al. [4], Da Silva et al. [24]. Let $G \in \mathcal{G}$ be a task parameter available to agent drawn from a distribution $P(G)$ where G defines the task and $\mathcal{G}$ is a set of all possible task parameters. Specifically, we define each task as a 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. The goal is to maximize the expected reward over the whole distribution of MDPs: $\int P(G)J(\pi, G)dG$, where $J(\pi, G) = \mathbb{E}_\pi[\sum_{t=0}^{T} \gamma^t r_t]$ is the expected return of the policy π given a task defined by G, γ is a discount factor, $\pi : \mathcal{S} \times \mathcal{G} \to \mathcal{A}$ is a multitask policy that we aim to learn, and r_t is the reward at time step t. 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 $\mathcal{G}_{test} \cap \mathcal{G}_{train} = \phi$.

3.2 Subtask Graph Execution Problem

The *subtask graph execution* problem is a multitask RL problem with a specific form of task parameter G called *subtask graph*. Figure 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.

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 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 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* ($\mathbf{o}_t \in \mathcal{O}$). Here, we assume that the agent is given a set of *options* ($\mathcal{O}$) [11, 25, 9] that performs subtasks by executing one or more primitive actions.

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 ($\mathcal{G}_{test}$).

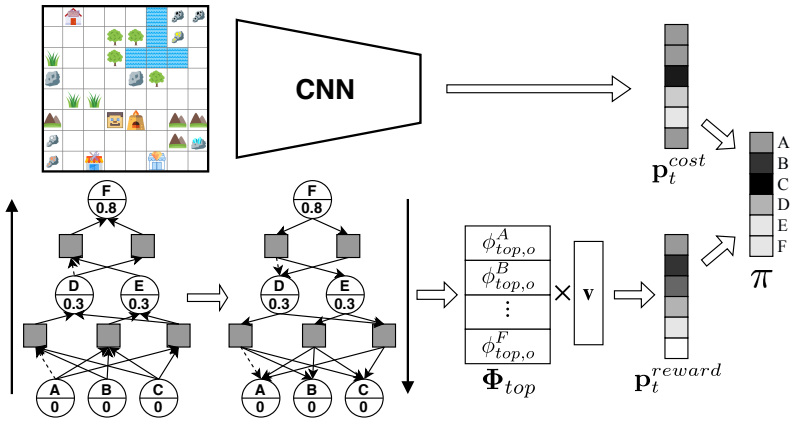

Figure 2: Neural subtask graph solver architecture. The task module encodes subtask graph through a bottom-up and top-down process, and outputs the reward score $\mathbf{p}_t^{reward}$. The observation module encodes observation using CNN and outputs the cost score $\mathbf{p}_t^{cost}$. The final policy is a softmax policy over the sum of two scores.

- **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 is not 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 since some objects are randomly moving. The agent needs to consider the expected outcome when deciding which subtask to execute.

4 Method

Our *neural subtask graph solver* (NSS) is a neural network which consists of a *task module* and an *observation module* as shown in Figure 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*) in section 3.2, learning to execute the subtask graph only from the reward is extremely challenging. To facilitate the learning, we propose *reward-propagation policy* (RProp), a non-parametric policy that propagates the reward information between related subtasks to model their dependencies. Since our RProp acts as a good initial policy, we train the NSS to approximate the RProp policy through policy distillation [26, 27], and finetune it through actor-critic method with generalized advantage estimation (GAE) [28] to maximize the reward. Section 4.1 describes the NSS architecture, and Section 4.2 describes how to construct the RProp policy.

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) [5] as follows:

$$\phi^i_{bot,o} = b_{\theta_o}\left(x^i_t, e^i_t, step, \sum_{j \in Child_i} \phi^j_{bot,a}\right), \qquad \phi^j_{bot,a} = b_{\theta_a}\left(\sum_{k \in Child_j} \left[\phi^k_{bot,o}, w^{j,k}_+\right]\right), \tag{1}$$

$$\phi^i_{top,o} = t_{\theta_o}\left(\phi^i_{bot,o}, r^i, \sum_{j \in Parent_i} \left[\phi^j_{top,a}, w^{i,j}_+\right]\right), \quad \phi^j_{top,a} = t_{\theta_a}\left(\phi^j_{bot,a}, \sum_{k \in Parent_j} \phi^k_{top,o}\right), \tag{2}$$

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 as 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. To sum up, the task module encodes the subtask graph using R3NN and estimates how good each subtask is.

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

$$\mathbf{p}^{cost}_t = \text{CNN}(\mathbf{obs}_t, step_t) \tag{3}$$

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 NSS policy is a softmax policy:

$$\pi(\mathbf{o}_t|\mathbf{s}_t, G) = \text{Softmax}(\mathbf{p}^{reward}_t + \mathbf{p}^{cost}_t), \tag{4}$$

which adds reward scores and cost scores.

4.2 Pre-training Neural Subtask Graph Solver from the Reward-Propagation Policy

Intuitively, the reward-propagation policy is designed to put high probabilities over subtasks that are likely to maximize the sum of *modified and smoothed* reward $\widehat{U}_t$ at time t, which will be defined in Eq. 9. Let $\mathbf{x}_t$ be a completion vector and $\mathbf{r}$ be a subtask reward vector (see Section 3 for definitions). Then, the sum of reward until time-step t is given as:

$$U_t = \mathbf{r}^T\mathbf{x}_t. \tag{5}$$

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:

$$\widetilde{U}_t = \mathbf{r}^T(\mathbf{x}_t + \mathbf{e}_t)/2. \tag{6}$$

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:

$$e^i_t = \underset{j\in Child_i}{\text{OR}}\left(y^j_{AND}\right), \quad y^j_{AND} = \underset{k\in Child_j}{\text{AND}}\left(\widehat{x}^{j,k}_t\right), \quad \widehat{x}^{j,k}_t = x^k_t w^{j,k} + (1-x^k_t)(1-w^{j,k}), \tag{7}$$

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, $\widehat{x}^{j,k}_t = 1$ 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 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:

$$\widetilde{e}^i_t = \underset{j\in Child_i}{\widetilde{\text{OR}}}\left(\widetilde{y}^j_{AND}\right), \quad \widetilde{y}^j_{AND} = \underset{k\in Child_j}{\widetilde{\text{AND}}}\left(\widehat{x}^{j,k}_t\right), \tag{8}$$

where $\widetilde{\text{AND}}$ and $\widetilde{\text{OR}}$ were implemented as scaled sigmoid and tanh functions as illustrated by Figure 3 (see appendix for details). With the smoothed operations, the sum of smoothed and modified reward is given as:

$$\widehat{U}_t = \mathbf{r}^T(\mathbf{x}_t + \widetilde{\mathbf{e}}_t)/2. \tag{9}$$

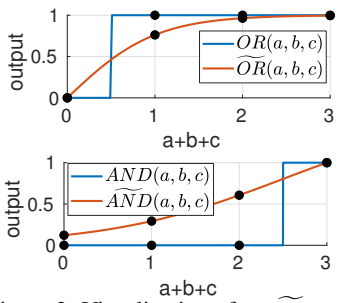

Figure 3: Visualization of OR, $\widetilde{\text{OR}}$, AND, and $\widetilde{\text{AND}}$ operations with three inputs (a,b,c). These smoothed functions are defined to handle arbitrary number of operands (see appendix).

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

$$\pi(\mathbf{o}_t|G, \mathbf{x}_t) = \text{Softmax}\left(\nabla_{\mathbf{x}_t}\widehat{U}_t\right) = \text{Softmax}\left(\frac{1}{2}\mathbf{r}^T + \frac{1}{2}\mathbf{r}^T\nabla_{\mathbf{x}_t}\widetilde{\mathbf{e}}_t\right), \tag{10}$$

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

4.3 Policy Optimization

The NSS architecture is first trained through policy distillation and finetuned using actor-critic method with generalized advantage estimation. During policy distillation, the KL divergence between NSS and teacher policy (RProp) is minimized as follows:

$$\nabla_\theta \mathcal{L}_1 = \mathbb{E}_{G\sim\mathcal{G}_{train}}\left[\mathbb{E}_{s\sim\pi_\theta^G}\left[\nabla_\theta D_{KL}\left(\pi_T^G||\pi_\theta^G\right)\right]\right], \tag{11}$$

where θ is the parameter of NSS architecture, π_θ^G is the simplified notation of NSS policy with subtask graph input G, π_T^G is the simplified notation of teacher (RProp) policy with subtask graph input G, $D_{KL}\left(\pi_T^G||\pi_\theta^G\right) = \sum_a \pi_T^G \log \frac{\pi_T^G}{\pi_\theta^G}$ and $\mathcal{G}_{train} \subset \mathcal{G}$ is the training set of subtask graphs. After policy distillation, we finetune NSS agent in an end-to-end manner using actor-critic method with generalized advantage estimation (GAE) [28] as follows:

$$\nabla_\theta \mathcal{L}_2 = \mathbb{E}_{G\sim\mathcal{G}_{train}}\left[\mathbb{E}_{s\sim\pi_\theta^G}\left[-\nabla_\theta \log \pi_\theta^G \sum_{l=0}^{\infty}\left(\prod_{n=0}^{l-1}(\gamma\lambda)^{k_n}\right)\delta_{t+l}\right]\right], \tag{12}$$

$$\delta_t = r_t + \gamma^{k_t} V_{\theta'}^\pi(\mathbf{s}_{t+1}, G) - V_{\theta'}^\pi(\mathbf{s}_t, G), \tag{13}$$

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 $\mathbb{E}\left[\left(R_t - V_{\theta'}^\pi(\mathbf{s}_t, G)\right)^2\right]$, where R_t is the discounted cumulative reward at time t. The complete procedure for training our NSS agent is summarized in Algorithm 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.

Algorithm 1 Policy optimization

$\mathcal{D} \leftarrow \emptyset$
while $|\mathcal{D}| < D$ **do**
  $G \sim \mathcal{G}_{train}$
  $d = \{(\mathbf{s}_t, \mathbf{o}_t, r_t, R_t), \ldots\} \sim \pi_\theta^G$ ▷ do rollout
  $\mathcal{D} \leftarrow \mathcal{D} \cup d$
for $d \in \mathcal{D}$ **do**
  if distillation **then**
    $\theta \leftarrow \theta + \eta_d \sum_d \nabla_\theta D_{KL}\left(\pi_T^G||\pi_\theta^G\right)$ ▷ update policy
  else if fine-tuning **then**
    Compute δ_t from Eq. 13 for all t
    $\theta \leftarrow \theta + \eta_{ac} \sum_d \nabla_\theta \log \pi_\theta^G \sum_{l=0}^{\infty}\left(\prod_{n=0}^{l-1}(\gamma\lambda)^{k_n}\right)\delta_{t+l}$ ▷ update policy
  $\theta' \leftarrow \theta' + \eta_c \sum_d \left(\nabla_{\theta'} V_{\theta'}^\pi(\mathbf{s}_t, G)\right)\left(R_t - V_{\theta'}^\pi(\mathbf{s}_t, G)\right)$ ▷ update critic

5 Experiment

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

5.1 Environment

We evaluated the performance of our agents on two domains: **Mining** and **Playground** that are developed based on MazeBase [29]. 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 RProp 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 1 and 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. The icons were downloaded from [30, 31]

Playground is a more flexible and challenging domain (see Figure 6). The subtask graph 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 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 [32], Minecraft [1] and XWORLD [2]. 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 would make other subtasks inexecutable.
- Delayed reward: Agent receives no reward from subtasks in the lower layers. But, the agent should execute some of them to make other subtasks eligible (see appendix for fully-delayed reward case).

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.
- **RProp** (Ours) is reward-propagation policy.
- **NSS** (Ours) is distilled from RProp policy and finetuned with actor-critic.
- **Independent** is an LSTM baseline trained on each subtask graph independently, similar to Independent model in [4]. It takes the same set of input as NSS except the subtask graph.

Task 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**
NSS (Ours)	**.820**	**.785**	**.715**	**.527**	**8.19**
RProp (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**
NSS (Ours)	**.828**	**.797**	**.733**	**.552**	**8.58**
Independent	.346	.296	.193	.188	3.89

Table 1: Generalization performance on unseen and larger subtask graphs. (Playground) **D1** consists of the same graph structure as training set, but the graph was unseen. **D2**, **D3**, and **D4** consist of (unseen) larger graph structures. (Mining) The subtask graphs in **Eval** are unseen during training.

To our best knowledge, previous hierarchical RL methods cannot directly generalize to unseen subtask graphs as they are not designed to take a task graph as input. Instead, we evaluated an instance of hierarchical RL method (**Independent** agent) in **adaptation** setting, as discussed in section 5.3.

5.3 Quantitative Result

Training Performance The learning curves of NSS agent and the performance of other agents are shown in Figure 4. Our RProp policy significantly outperforms the Greedy policy. This implies that the proposed idea of back-propagating the reward gradient captures long-term dependencies among subtasks to some extent. We also found that NSS further improves the performance through fine-tuning with actor-critic method. We hypothesize that NSS learned to estimate the expected costs of executing subtasks from the observations and consider them along with subtask graphs.

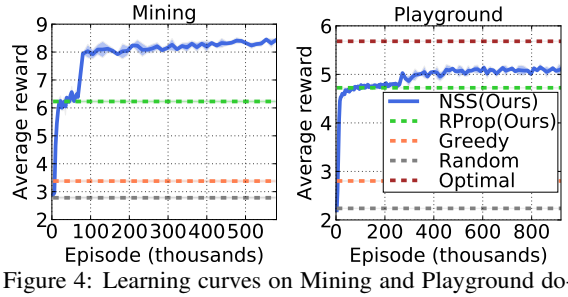

Figure 4: Learning curves on Mining and Playground domain. NSS is distilled from RProp on 77K and 256K episodes, respectively, and finetuned after that.

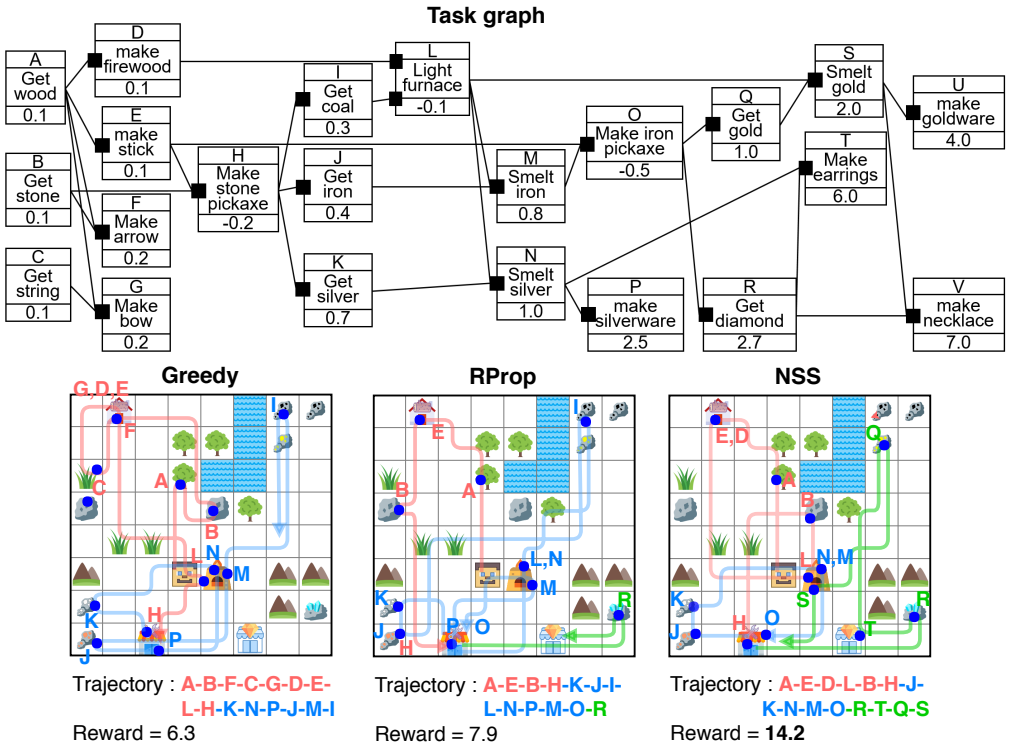

Figure 5: Example trajectories of Greedy, RProp, and NSS agents given 75 steps on Mining domain. We used different colors to indicate that agent has different types of pickaxes: red (no pickaxe), blue (stone pickaxe), and green (iron pickaxe). Greedy agent prefers subtasks C, D, F, and G to H and L since C, D, F, and G gives positive immediate reward, whereas NSS and RProp agents find the shortest path to make stone pickaxe, and focus on subtasks with a higher reward. Compared to RProp, NSS agent can find a shorter path to make an iron pickaxe, and succeeds to execute more number of subtasks.

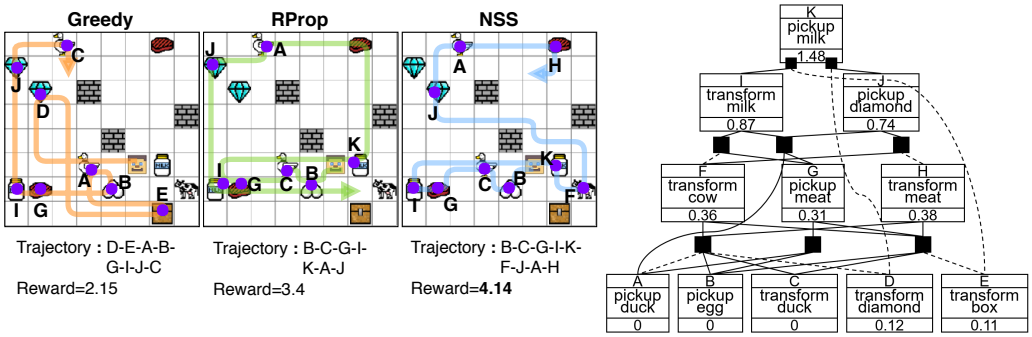

Figure 6: Example trajectories of Greedy, RProp, and NSS agents given 45 steps on Playground domain. The subtask graph includes NOT operation and distractor (subtask D, E, and H). We removed stochasticity in environment for the controlled experiment. Greedy agent executes the distractors since they give positive immediate rewards, which makes it impossible to execute the subtask K which gives the largest reward. RProp and NSS agents avoid distractors and successfully execute subtask K by satisfying its preconditions. After executing subtask K, NSS agent found a shorter path to execute remaining subtasks than RProp, and get larger reward.

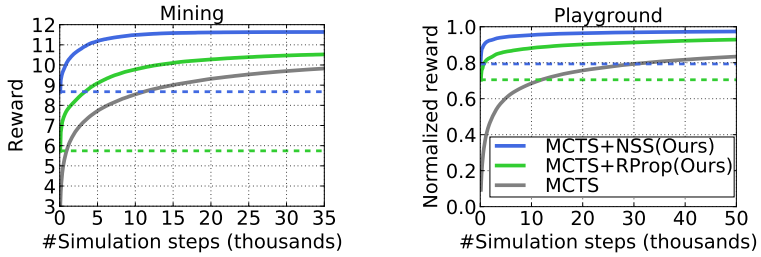

Figure 7: Performance of MCTS+NSS, MCTS+RProp and MCTS per the number of simulated steps on (Left) **Eval** of Mining domain and (Right) **D2** of Playground domain (see Table 1).

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 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, NSS 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, NSS 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 NSS.

5.4 Qualitative Result

Figure 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, NSS and RProp agents focus on executing subtask H (make stone pickaxe) in order to collect materials much faster in the long run. Compared to RProp, NSS learns to consider observation also and avoids subtasks with high cost (e.g., get coal).
Figure 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 agent chooses to execute distractors, since they gives positive reward while subtasks A, B, and C does not. However, RProp agent observes non-zero gradient for subtasks A, B, and C that are propagated from the parent nodes. Thus, even though the reward is delayed, RProp can figure out which subtask to execute. NSS learns to understand long-term dependencies from RProp, and finds shorter path by also considering the observation. For the fully-delayed reward case, please refer the appendix.

5.5 Combining NSS with Monte-Carlo Tree Search

We further investigated how well our NSS agent performs when the simulation model is available to the agent. To this end, we compared our NSS agent to conventional search-based methods and showed how our NSS agent can be combined with search-based methods to further improve the performance.

We implemented the following methods (see appendix for the detail):

- MCTS: An MCTS algorithm with UCB [33] criterion for choosing actions.
- MCTS+NSS: An MCTS algorithm combined with our NSS agent. NSS policy was used as a rollout policy to explore reasonably good states during tree search, which is similar to AlphaGo [34].
- MCTS+RProp: An MCTS algorithm combined with our RProp agent similar to MCTS+NSS.

The results are shown in Figure 7. It turns out that our NSS performs as well as MCTS method with approximately 32K simulations on Playground and 11K simulations on Mining domain. This indicates that our NSS agent implicitly performs long-term reasoning that is not easily achievable by a sophisticated MCTS, even though NSS does not use any simulation and has never seen such subtask graphs during training. More interestingly, MCTS+NSS and MCTS+RProp significantly outperforms MCTS, and MCTS+NSS 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+NSS performs almost as well as the Optimal policy with only 0.005% simulations compared to the Optimal policy. This result implies that NSS can also be used to improve simulation-based planning methods by effectively reducing the search space.

6 Conclusion

We introduced the subtask graph execution problem which is an effective and principled way of describing complex tasks. To address the difficulty of dealing with complex subtask dependencies, we proposed a 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 is given. However, the future work might extend to the case where the subtask graph is unknown. This setting is extremely challenging for complex task dependencies, but we hypothesize that subtask graph might be also learnable through experience.

References

[1] Junhyuk Oh, Satinder Singh, Honglak Lee, and Pushmeet Kohli. Zero-shot task generalization with multi-task deep reinforcement learning. *arXiv preprint arXiv:1706.05064*, 2017.

[2] 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.

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

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

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

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

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

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

[9] 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, 1999.

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

[11] Doina Precup. *Temporal abstraction in reinforcement learning*. University of Massachusetts Amherst, 2000.

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

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

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

[28] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

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

[30] `https://icons8.com/`, .

[31] `https://www.flaticon.com/`, .

[32] 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.

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

[34] 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.

A Details of the Task

We define each task as a 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 I 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:

$$x_{t+1}^i = \begin{cases} 1 & \text{if} \quad e_t^i = 1 \\ x_t^i & \text{otherwise} \end{cases}. \tag{14}$$

The observation is updated such that agent moves on to the target object, and perform corresnponding primitive action (See Section H 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:

$$e_{t+1}^i = \underset{j \in Child_i}{\text{OR}} \left(y_{AND}^j \right), \tag{15}$$

$$y_{AND}^i = \underset{j \in Child_i}{\text{AND}} \left(\widehat{x}_{t+1}^{i,j} \right), \tag{16}$$

$$\widehat{x}_{t+1}^{i,j} = x_{t+1}^j w^{i,j} + (1 - x_{t+1}^j)(1 - w^{i,j}), \tag{17}$$

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, $\widehat{x}_t^{i,j} = 1$ 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. 17. Specifically, when agent executes subtask i, the reward given to agent at time step t is given as follows:

$$r_t = \begin{cases} r^i & \text{if} \quad e_t^i = 1 \\ 0 & \text{otherwise} \end{cases}. \tag{18}$$

B Details of NSS Architecture

Task module Figure 8 illustrates the structure of the task module of NSS architecture for a given input subtask graph. Specifically, the task module was implemented with four encoders: $b_{\theta_a}, b_{\theta_o}, t_{\theta_a}$,

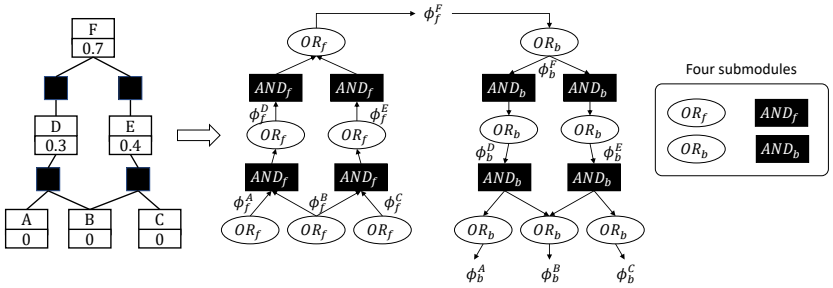

Figure 8: An example of R3NN construction for a given subtask graph input. The four encoders ($b_{\theta_a}, b_{\theta_o}, t_{\theta_a}$, and t_{θ_o}) are cloned and connected according to the input subtask graph where the cloned models share the weight. For simplicity, only the output embeddings of bottom-up and top-down `OR` encoder were specified in the figure.

and t_{θ_o}. The input and output of each encoder is defined in the main text section **4.1** as:

$$\phi^i_{bot,o} = b_{\theta_o}\left(x^i_t, e^i_t, step, \sum_{j \in Child_i} \phi^j_{bot,a}\right), \qquad \phi^j_{bot,a} = b_{\theta_a}\left(\sum_{k \in Child_j} \left[\phi^k_{bot,o}, w^{j,k}_+\right]\right), \tag{19}$$

$$\phi^i_{top,o} = t_{\theta_o}\left(\phi^i_{bot,o}, r^i, \sum_{j \in Parent_i} \left[\phi^j_{top,a}, w^{i,j}_+\right]\right), \quad \phi^j_{top,a} = t_{\theta_a}\left(\phi^j_{bot,a}, \sum_{k \in Parent_j} \phi^k_{top,o}\right), \tag{20}$$

For bottom-up process, the encoder takes the output embeddings of its children encoders as input. Similarly, for top-down process, the encoder takes the output embeddings of its parent encoders as input. The input embeddings are aggregated by taking element-wise summation. For $\phi^i_{bot,a}$ and $\phi^i_{top,o}$, the embeddings are concatenated with $w^{i,j}_+$ to deal with `NOT` connection before taking the element-wise summation. Then, the summed embedding is concatenated with all additional input as defined in Eq. 19 and 20, which is further transformed with three fully-connected layers with 128 units. The last fully-connected layer outputs 128-dimensional output embedding. The embeddings are transformed to reward scores as 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. Similarly, the reward baseline is computed by $b^{reward}_t = \text{sum}(\mathbf{\Phi}^\top_{top}\tilde{\mathbf{v}})$, where sum$(\cdot)$ is the reduced-sum operation and $\tilde{\mathbf{v}}$ is the weight vector for reward baseline. We used parametric ReLU (PReLU) function as activation function.

Observation module The network consists of BN1-Conv1(16x1x1-1/0)-BN2-Conv2(32x3x3-1/1)-BN3-Conv3(64x3x3-1/1)-BN4-Conv4(96x3x3-1/1)-BN5-Conv5(128x3x3-1/1)-BN6-Conv6(64x1x1-1/0)-FC(256). The output embedding of FC(256) was then concatenated with the number of remaining time step $step_t$. Finally, the network has two fully-connected output layers for the cost score $\mathbf{p}^{cost}_t \in \mathbb{R}^N$ and the cost baseline $b^{cost}_t \in \mathbb{R}$. Then, the policy of NSS is calculated by adding reward score and cost score, and taking softmax:

$$\pi(\mathbf{o}_t|\mathbf{s}_t, G) = \text{Softmax}(\mathbf{p}^{reward}_t + \mathbf{p}^{cost}_t). \tag{21}$$

The baseline output is obtained by adding reward baseline and cost baseline:

$$V_{\theta'}(\mathbf{s}_t, G) = b^{reward}_t + b^{cost}_t. \tag{22}$$

C Details of Learning NSS Agent

Learning objectives The NSS architecture is first trained through policy distillation and finetuned using actor-critic method with generalized advantage estimator. During policy distillation, the KL divergence between NSS and teacher policy (RProp) is minimized as follows:

$$\nabla_\theta \mathcal{L}_1 = \mathbb{E}_{G \sim \mathcal{G}_{train}}\left[\mathbb{E}_{s \sim \pi^G_\theta}\left[\nabla_\theta D_{KL}\left(\pi^G_T || \pi^G_\theta\right)\right]\right], \tag{23}$$

where θ is the parameter of NSS architecture, π_θ^G is the simplified notation of NSS policy with subtask graph input G, π_T^G is the simplified notation of teacher (RProp) policy with subtask graph input G, $D_{KL}\left(\pi_T^G||\pi_\theta^G\right) = \sum_a \pi_T^G \log \frac{\pi_T^G}{\pi_\theta^G}$ and $\mathcal{G}_{train} \subset \mathcal{G}$ is the training set of subtask graphs.

For both policy distillation and fine-tuning, we sampled one subtask graph for each 16 parallel workers, and each worker in turn sample a mini-batch of 16 world configurations (maps). Then, NSS generates total 256 episodes in parallel. After generating episode, the gradient from 256 episodes are collected and averaged, and then back-propagated to update the parameter. For policy distillation, we trained NSS for 40 epochs where each epoch involves 100 times of update. Since our RProp policy observes only the subtask graph, we only trained task module during policy distillation. The observation module was trained for auxiliary prediction task; observation module predicts the number of step taken by agent to execute each subtask.

After policy distillation, we finetune NSS agent in an end-to-end manner using actor-critic method with generalized advantage estimation (GAE) [28] as follows:

$$\nabla_\theta \mathcal{L}_2 = \mathbb{E}_{G\sim\mathcal{G}_{train}}\left[\mathbb{E}_{s\sim\pi_\theta^G}\left[-\nabla_\theta \log \pi_\theta^G \sum_{l=0}^{\infty}\left(\prod_{n=0}^{l-1}(\gamma\lambda)^{k_n}\right)\delta_{t+l}\right]\right], \tag{24}$$

$$\delta_t = r_t + \gamma^{k_t} V_{\theta'}^\pi(\mathbf{s}_{t+1}, G) - V_{\theta'}^\pi(\mathbf{s}_t, G), \tag{25}$$

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 $\mathbb{E}\left[\left(R_t - V_{\theta'}^\pi(\mathbf{s}_t, G)\right)^2\right]$, where R_t is the discounted cumulative reward at time t.

Hyperparameters For both finetuning and policy distillation, we used RMSProp optimizer with the smoothing parameter of 0.97 and epsilon of 1e-6. When distilling agent with teacher policy, we used learning rate=1e-4 and multiplied it by 0.97 on every epoch for both Mining and Playground domain. For finetuning, we used learning rate=2.5e-6 for Playground domain, and 2e-7 for Mining domain. For actor-critic training for NSS, we used $\alpha = 0.03,\ \lambda = 0.96,\ \gamma = 0.99$.

D Details of AND/OR Operation and Approximated AND/OR Operation

In section 4.2, the output of i-th AND and OR node in subtask graph were defined using AND and OR operation with multiple input. They can be represented in logical expression as below:

$$\underset{j\in Child_i}{\text{OR}}\left(y^j\right) = y^{j_1} \vee y^{j_2} \vee \ldots \vee y^{j_{|Child_i|}}, \tag{26}$$

$$\underset{j\in Child_i}{\text{AND}}\left(y^j\right) = y^{j_1} \wedge y^{j_2} \wedge \ldots \wedge y^{j_{|Child_i|}}, \tag{27}$$

where $j_1, \ldots, j_{|Child_i|}$ are the elements of a set $Child_i$ and $Child_i$ is the set of inputs coming from the children nodes of i-th node. Then, these AND and OR operations are smoothed as below:

$$\widetilde{\underset{j\in Child_i}{\text{OR}}}\left(\widetilde{y}_{AND}^j\right) = h_{or}\left(\sum_{j\in Child_i}\widetilde{y}_{AND}^j\right), \tag{28}$$

$$\widetilde{\underset{j\in Child_i}{\text{AND}}}\left(\widehat{x}_t^{i,j}\right) = h_{and}\left(\sum_{j\in Child_i}\widehat{x}_t^{i,j} - |Child_i| + 0.5\right), \tag{29}$$

where $h_{or}(x) = \alpha_o \text{tanh}(x/\beta_o)$, $h_{and}(x) = \alpha_a \sigma(x/\beta_a)$, $\sigma(\cdot)$ is sigmoid function, and $\alpha_o, \beta_o, \alpha_a, \beta_a \in \mathbb{R}$ are hyperparameters to be set. We used $\beta_a = 0.6, \beta_o = 2, \alpha_a = 1/\sigma(0.25), \alpha_o = 1$ for Mining domain, and $\beta_a = 0.5, \beta_o = 1.5, \alpha_a = 1/\sigma(0.25), \alpha_o = 1$ for Playground domain.

E Details of Subtask Executor

Architecture The subtask executor has the same architecture of the parameterized skill architecture of [1] with slightly different hyperparameters. The network consists of Conv1(32x3x3-

1/1)-Conv2(32x3x3-1/1)-Conv3(32x1x1-1/0)-Conv4(32x3x3-1/1)-LSTM(256)-FC(256). The subtask executor takes two task parameters ($q = [q^{(1)}, q^{(2)}]$) as additional input and computes $\chi(q) = ReLU(W^{(1)}q^{(1)} \odot W^{(2)}q^{(2)})$ to compute the subtask embedding, and further linearly transformed into the weights of Conv3 and the (factorized) weight of LSTM through multiplicative interaction as described above. Finally, the network has three fully-connected output layers for actions, termination probability, and baseline, respectively.

Learning objective The subtask executor is trained through policy distillation and then finetuned. Similar to [1], we first trained 16 teacher policy network for each subtask. The teacher policy network consists of Conv1(16x3x3-1/1)-BN1(16)-Conv2(16x3x3-1/1)-BN2(16)-Conv3(16x3x3-1/1)-BN3(16)-LSTM(128)-FC(128). Similar to subtask executor network, the teacher policy network has three fully-connected output layers for actions, termination probability, and baseline, respectively. Then, the learned teacher policy networks are used as teacher policy for policy distillation to train subtask executor. During policy distillation, we train agent to minimize the following objective function:

$$\nabla_\xi \mathcal{L}_{1,sub} = \mathbb{E}_{\mathbf{o}\sim\mathcal{O}}\left[\mathbb{E}_{s\sim\pi^{\mathbf{o}}_\xi}\left[\nabla_\xi\left\{D_{KL}\left(\pi^{\mathbf{o}}_T||\pi^{\mathbf{o}}_\xi\right)\right] + \alpha L_{term}\right\}\right], \tag{30}$$

where ξ is the parameter of subtask executor network, $\pi^{\mathbf{o}}_\xi$ is the simplified notation of subtask executor given input subtask $\mathbf{o}$, $\pi^{\mathbf{o}}_T$ is the simplified notation of teacher policy for subtask $\mathbf{o}$, $L_{term} = -\mathbb{E}_{\mathbf{s}_t\in\tau_{\mathbf{o}}}\left[\log\beta_\xi(\mathbf{s}_t,\mathbf{o})\right]$ is the cross entropy loss of predicting termination, $\tau_{\mathbf{o}}$ is a set of state in which the subtask $\mathbf{o}$ is terminated, $\beta_\xi(s_t,\mathbf{o})$ is the termination probability output, and $D_{KL}\left(\pi^{\mathbf{o}}_T||\pi^{\mathbf{o}}_\xi\right) = \sum_a \pi^{\mathbf{o}}_T(a|s)\log\frac{\pi^{\mathbf{o}}_T(a|s)}{\pi^{\mathbf{o}}_\xi(a|s)}$. After policy distillation, we finetuned subtask executor using actor-critic method with generalized advantage estimation (GAE):

$$\nabla_\xi \mathcal{L}_{2,sub} = \mathbb{E}_{\mathbf{o}\sim\mathcal{O}}\left[\mathbb{E}_{s\sim\pi^{\mathbf{o}}_\xi}\left[-\nabla_\xi\log\pi_\xi\left(\mathbf{a}_t|\mathbf{obs}_t,\mathbf{o}\right)\sum_{k=0}^{\infty}(\gamma\lambda)^k\delta_{t+k} + \alpha\nabla_\xi L_{term}\right]\right], \tag{31}$$

where $\gamma\in[0,1]$ is a discount factor, $\lambda\in[0,1]$ is a weight for balancing between bias and variance of the advantage estimation, and $\delta_t = r_t + \gamma V^\pi(\mathbf{obs}_{t+1};\xi') - V^\pi(\mathbf{obs}_t;\xi')$. We used $\lambda = 0.96$, $\gamma = 0.99$ for fine-tuning, and $\alpha = 0.1$ for both policy distillation and fine-tuning.

F Details of LSTM Baseline

Architecture The LSTM baseline consists of LSTM on top of CNN. The architecture of CNN is the same as the CNN architecture of observation module of NSS described in the section B, and the architecture of LSTM is the same as the LSTM architecture used in subtask executor described in the section E. Specifically, it consists of BN1-Conv1(16x1x1-1/0)-BN2-Conv2(32x3x3-1/1)-BN3-Conv3(64x3x3-1/1)-BN4-Conv4(96x3x3-1/1)-BN5-Conv5(128x3x3-1/1)-BN6-Conv6(64x1x1-1/0)-LSTM(256)-FC(256). The CNN takes the observation tensor as an input and outputs an embedding. The embedding is then concatenated with other input vectors including subtask completion indicator $\mathbf{x}_t$, eligibility vector $\mathbf{e}_t$, and the remaining step $step_t$. Finally, LSTM takes the concatenated vector as an input and output the softmax policy with the parameter θ': $\pi_{\theta'}\left(\mathbf{o}_t|\mathbf{obs}_t,\mathbf{x}_t,\mathbf{e}_t,step_t\right)$.

Learning objective The LSTM baseline was trained using actor-critic method. We found that the moving average of cumulative reward works better than learning critic network, and used it for computing the advantage. Since the subtask graph is fixed in adaptation setting, the moving average of the cumulative reward works well as a value function. In fact, we found that the moving average of the cumulative reward works much better than learning critic network as a value function. Similar to NSS, the learning objective is given as

$$\nabla_{\theta'}\mathcal{L}_{LSTM} = \mathbb{E}_{s\sim\pi^G_{\theta'}}\left[-\nabla_{\theta'}\log\pi_{\theta'}\left(\mathbf{o}_t|\mathbf{obs}_t,\mathbf{x}_t,\mathbf{e}_t,step_t\right)\sum_{l=0}^{\infty}\left(\prod_{n=0}^{l-1}(\gamma\lambda)^{k_n}\right)\delta_{t+l}\right], \tag{32}$$

where $\gamma\in[0,1]$ is a discount factor, $\lambda\in[0,1]$ is a weight for balancing between bias and variance of the advantage estimation, $\delta_t = r_t + \gamma^{k_t}\overline{V}(t+1) - \overline{V}(t)$, and $\overline{V}(t)$ is the moving average of cumulative reward at time step t. We used $\lambda = 0.96$ and $\gamma = 0.99$.

G Details of Search Algorithms

Each iteration of Monte-Carlo tree search method consists of four stages: selection-expansion-rollout-backpropagation. For selection, we used UCB criterion [[33]]. Specifically, the option for which the score below has the highest value is chosen for selection:

$$\text{score} = \frac{R_i}{n_i} + C_{UCB}\sqrt{\frac{\ln N}{n_i}}, \tag{33}$$

where R_i is the accumulated return at i-th node, n_i is the number of visit of i-th node, C_{UCB} is the exploration-exploitation balancing weight, and N is the number of total iterations so far. We found that $C_{UCB} = 2\sqrt{2}$ gives the best result and used it for MCTS, MCTS+RProp and MCTS+NSS methods. For expansion, MCTS randomly chooses the remaining eligible subtask, while the subtask is chosen by NSS policy for MCTS+NSS method and RProp policy for MTS+RProp method. More specifically, the subtask for which the NSS and RProp policy probability have the highest value is chosen for MCTS+NSS and MCTS+RProp method, respectively. Due to a memory limit, the expansion of search tree was truncated at the depth of 7 for Playground and 10 for Mining domains, and performed rollout after the maximum depth. In rollout, MCTS randomly executes an eligible subtask, while MCTS+NSS and MCTS+RProp execute again the subtask with the highest probability given by NSS and RProp policies, respectively. Once the episode is terminated, the result is back-propagated; the estimated value function and visit count are updated for each node in the tree.

H Details of Environment

H.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.

H.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 I for details).

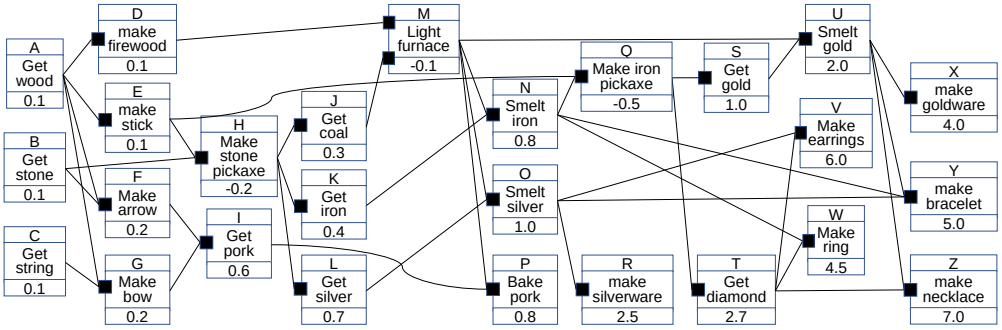

Figure 9: The entire graph of Mining domain. Based on this graph, we generated 640 subtask graphs by removing the subtask node that has no parent node.

I Details of Subtask Graph Generation

I.1 Mining Domain

The precondition of each subtask in Mining domain was defined as Figure 9. 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 $0.8 \sim 1.2$.

I.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 2: 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 2. 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 3 and 5, while N_T and N_D was manually set. We only generated a graph without duplicated AND nodes with same children node. We carefully set the range of each parameter such that at least 500 different subtask graph can be generated with given parameters. The table 3 and 5 summarizes parameters used to generate training and evaluation subtask graphs.

J Ablation Study on Neural Subtask Graph Solver Agent

J.1 Learning without Pre-training

We implemented **NSS-scratch** agent that is trained with actor-critic method from scratch without pre-training from RProp policy to show that pre-training plays a crucial role for training our NSS agent. Table 4 summarizes the result. NSS-scratch performs much worse than NSS, suggesting that pre-training is important in training NSS. This is not surprising as our problem is combinatorially intractable (e.g. searching over optimal sequence of subtasks given an unseen subtask graph).

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 3: Task graph parameters for training set and tasks **D1**$\sim$**D4**.

Zero-Shot Performance					
	Playground($\bar{R}$)				Mining(R)
Task	**D1**	**D2**	**D3**	**D4**	**Eval**
NSS (Ours)	**.820**	**.785**	**.715**	**.527**	**8.19**
NSS-task (Ours)	.773	.730	.645	.387	6.51
RProp (Ours)	.721	.682	.623	.424	6.16
NSS-scratch (Ours)	.046	.056	.062	.106	3.68
Random	0	0	0	0	2.79

Table 4: Zero-shot generalization performance on Playground and Mining domain. NSS-scratch agent performs much worse than NSS and RProp agent on Playground and Mining domain.

J.2 Ablation Study on the Balance between Task and Observation Module

We implemented **NSS-task** agent that uses only the task module without observation module to compare the contribution of task module and observation module of NSS agent. Overall, our NSS agent outperforms the NSS-task agent, showing that the observation module improves the performance by a large margin.

K Experiment Result on Subtask Graph Features

To investigate how agents deal with different types of subtask graph components, we evaluated all

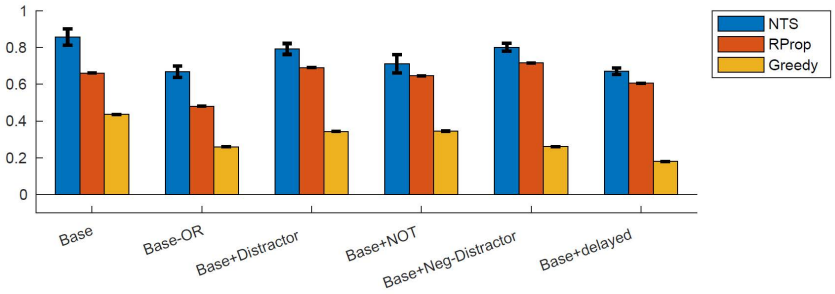

Figure 10: Normalized performance on subtask graphs with different types of dependencies.

agents on the following types of subtask graphs:

- 'Base' set consists of subtask graphs with only `AND` and `OR` operation.
- 'Base-OR' set removes all the `OR` operations from the base set.
- 'Base+Distractor' set adds several distractor subtasks to the base set.
- 'Base+NOT' set adds several `NOT` operations to the base set.
- 'Base+NegDistractor' set adds several negative distractor subtasks to the base set.
- 'Base+Delayed' set assigns zero reward to all subtasks but the top-layer subtask.

Note that we further divided the set of Distractor into Distractor and NegDistractor. The distractor subtask is a subtask without any parent node in the subtask graph. Executing this kind of subtask may give an immediate reward but is sub-optimal in the long run. The negative-distractor subtask is a subtask with only and at least one `NOT` connection to parent nodes in the subtask graph. Executing this subtask may give an immediate reward, but this would make other subtasks not executable. Table 5 summarizes the detailed parameters used for generating subtask graphs. The results are shown in Figure 10. Since 'Base' and 'Base-OR' sets do not contain `NOT` operation and every subtask gives a positive reward, the greedy baseline performs reasonably well compared to other sets of subtask graphs. It is also shown that the gap between NSS and RProp is relatively large in these two sets. This is because computing the optimal ordering between subtasks is more important in these kinds of subtask graphs. Since only NSS can take into account the cost of each subtask from the observation, it can find a better sequence of subtasks more often.

In 'Base+Distractor', 'Base+NOT', and 'Base+NegDistractor' cases, it is more important for the agent to carefully find and execute subtasks that have a positive effect in the long run while avoiding distractors that are not helpful for executing future subtasks. In these tasks, the greedy baseline tends to execute distractors very often because it cannot consider the long-term effect of each subtask in principle. On the other hand, our RProp can naturally screen out distractors by getting zero or negative gradient during reward back-propagation. Similarly, RProp performs well on 'Base+Delayed' set because it gets non-zero gradients for all subtasks that are connected to the final rewarding subtask. Since our NSS was distilled from RProp, it can handle delayed reward or distractors as well as (or better than) RProp.

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 5: Subtask graph parameters for analysis of subtask graph components.