

國立臺灣大學電機資訊學院電機工程學系  
碩士論文



Department of Electrical Engineering  
College of Electrical Engineering and Computer Science  
National Taiwan University  
Master Thesis

利用遺傳程式設計從專家示範自動推論任務子結構  
Automatic Induction of Task Substructures from Expert  
Demonstrations via Genetic Programming

劉容均  
Jung-Chun Liu

指導教授：于天立 博士  
Advisor: Tian-Li Yu, Ph.D.

中華民國 112 年 7 月  
July, 2023



# Abstract

To deal with hierarchical and compositional decision-making problems, intelligent agents necessitate domain knowledge representation on task structures and subtask rules for planning and reasoning. Previous approaches often rely on strong assumptions about pre-defined subtasks due to the difficulty of determining subtasks lacking domain knowledge. Therefore, we propose a framework that automatically induces subtasks from expert demonstrations to solve complex tasks. The framework encompasses planning, deep reinforcement learning (DRL), and evolutionary computation, and the procedure involves inducing symbolic rules, constructing task structures from goals, and providing intrinsic rewards based on task structures. We utilize genetic programming for symbolic rule induction, where the selection of the rule model reflects prior domain knowledge of effect rules. We evaluate the framework in two environments, including the MINECRAFT environment, and demonstrate that it improves the performance of DRL agents. In addition, we also demonstrate the generalizability in task and skill level by composing the task structure and inducing the new rules. This research contributes insights into integrated frameworks as a cognitive architecture to address hierarchical real-world problems.

## Keywords

Inductive Learning, Learning from Demonstration, Decision Making, Deep Reinforcement Learning, Classical Planning, Genetic Programming



# Contents

<b>Abstract</b>	<b>i</b>
<b>Contents</b>	<b>ii</b>
<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>ix</b>
<b>List of Symbols</b>	<b>x</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>5</b>
2.1 Deep Reinforcement Learning . . . . .	6
2.2 Classical Planning . . . . .	6
2.3 Evolutionary Computation . . . . .	7
2.4 Related Works . . . . .	8
2.4.1 Learning Abstraction from Demonstrations . . . . .	8
2.4.2 Subtask Inference . . . . .	9
<b>3 Problem Formulation and Preliminaries</b>	<b>11</b>
3.1 Markov Decision Process . . . . .	11
3.2 Planning Domain Definition Language . . . . .	12
3.3 MDP Problems in Planning Language . . . . .	13

3.4	Critical Action . . . . .	15
3.4.1	Effect and Precondition Variable Space . . . . .	16
3.4.2	Action Schema . . . . .	17
3.4.3	Effect Rule . . . . .	19
3.4.4	Precondition Rule . . . . .	20
3.5	Critical Action Graph . . . . .	20
<b>4</b>	<b>Inducing Hierarchical Structure and Symbolic Knowledge</b>	<b>23</b>
4.1	Overview . . . . .	23
4.2	Induction Module . . . . .	24
4.2.1	Collecting Demonstrations . . . . .	25
4.2.2	Extracting Action-Effect Linkages from Demonstrations . . . . .	26
4.2.3	Determining Effect Symbolic Rules . . . . .	28
4.2.4	Determining Precondition . . . . .	29
4.2.5	Summary . . . . .	31
4.3	Training Module . . . . .	32
4.3.1	Inferring Critical Action Graph . . . . .	33
4.3.2	Deep Reinforcement Learning Agent . . . . .	33
4.3.3	Intrinsic Rewards . . . . .	34
4.3.4	Summary . . . . .	35
<b>5</b>	<b>Test Environments and Experiment Settings</b>	<b>37</b>
5.1	SWITCH . . . . .	38
5.1.1	Number of Switches . . . . .	40
5.1.2	Order . . . . .	40
5.1.3	Distractors . . . . .	41
5.1.4	Four Rooms . . . . .	41
5.2	Minecraft . . . . .	42

5.3	Implementation Detail . . . . .	45
5.3.1	Advantage Actor-Critic . . . . .	46
5.3.2	Genetic Programming . . . . .	47
5.3.3	Reward Function . . . . .	50
5.4	Compared Algorithms . . . . .	50
5.4.1	Proximal Policy Optimization and Deep Q-Network . . . . .	51
5.4.2	Rewarding Impact-Driven Exploration . . . . .	51
5.4.3	Behavior Cloning . . . . .	52
5.4.4	Generative Adversarial Imitation Learning . . . . .	53
<b>6</b>	<b>Experiments and Discussions</b>	<b>54</b>
6.1	Preliminary Experiment Results . . . . .	54
6.1.1	Action-Effect Linkage . . . . .	55
6.1.2	Symbolic Regression . . . . .	55
6.1.3	Intrinsic Reward and Pre-Training . . . . .	57
6.2	Experiment Results on Performance . . . . .	60
6.2.1	SWITCH . . . . .	60
6.2.2	MINECRAFT . . . . .	62
6.3	Generalizability . . . . .	63
6.3.1	Preliminaries . . . . .	64
6.3.2	Task Generalization . . . . .	65
6.3.3	Skill Generalization . . . . .	66
6.4	Discussions . . . . .	67
6.4.1	Extracting Task Structures from Demonstrations . . . . .	67
6.4.2	Comparison with Other Methods . . . . .	67
6.4.3	Limitation . . . . .	69
<b>7</b>	<b>Conclusion and Future Work</b>	<b>72</b>

## **Bibliography**

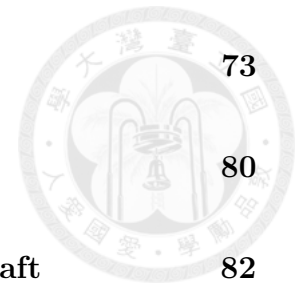
### **A Task Structure of Test Problems**

### **B Mutual Information of Action-Effect Pairs in Minecraft**

**73**

**80**

**82**





# List of Figures

3.1	The illustration of reinforcement learning in MDP problems. The agent observes the state $s$ and executes the action $a$ ; then the environment gives the agent reward $R(s, a)$ and the next state $s' = T(s, a)$ based on $s$ and $a$ . . . . .	12
3.2	An simplified example in MINECRAFT. . . . .	14
3.3	The illustration of <code>make_stick</code> action schema. . . . .	18
3.4	The critical action graph for <code>make_stick</code> . . . . .	21
4.1	The overview of the framework. The induction module infers the critical actions online, and the training module builds critical action graphs and provides intrinsic rewards for agents. . . . .	24
4.2	The procedure of induction modules. . . . .	25
4.3	The illustration of symbolic regression using genetic programming. . . . .	29
4.4	The workflow of the induction module with input and output of each method. . . . .	32
4.5	The workflow of the training module with input and output of each method. . . . .	35
5.1	Visualization of SWITCH environment. The goal of the agent is to turn on all switches in a specific order. The figure shows the state of 4-SWITCHES-INCREMENTAL. Green circles indicate <i>on</i> , yellow switches indicate <i>available</i> , and gray switches indicate <i>off</i> . . . . .	39
5.2	Visualization of SWITCH environment with 4, 8 and 16 switches. . . . .	41

5.3	Visualization of SWITCH environment with distractors in different order. The dotted stroke circles represent distractors which is initially available. . . . .	42
5.4	Visualization of SWITCH environment with four rooms. The gray squares represent walls which can not move across. . . . .	42
5.5	Visualization of MINECRAFT environment. . . . .	43
5.6	Dependency of subtasks in MINECRAFT. We select IRON and ENHANCETABLE as test problems, and the detail illustration about effects and the preconditions of the problems are shown in Appendix A. . . . .	45
5.7	The architecture of DRL agents incorporates A2C methods. The actor generates actions, and the critic evaluates the actor's output. . . . .	46
5.8	The procedure of genetic programming. . . . .	48
6.1	Heatmap of mutual information. . . . .	56
6.2	The accuracy of symbolic regression using genetic programming. The demonstrations are sampled from random subtasks, and the accuracy is the success rate of inducing 27 rules in MINECRAFT out of five runs. . . . .	57
6.3	The training performance with intrinsic rewards in 4-SWITCHES environment. The baseline is A2C without intrinsic rewards. . . . .	59
6.4	The training performance comparing pretraining and intrinsic rewards in 16-SWITCHES environment. . . . .	59
6.5	Training performance in SWITCH with different numbers of switches. . . . .	61
6.6	Training performance in SWITCH with distractors and rooms. . . . .	61
6.7	Training performance in MINECRAFT environment. . . . .	63
6.8	The performance of BC, GAIL and our methods with different demonstrations after 5M steps in 4-SWITCHES-INCREMENTAL. . . . .	64
6.9	The illustration of inducing task structures from critical action sequences. . . . .	68



6.10	The training performance of our framework in MINECRAFT-MULTIPLE with the deterministic and probabilistic environments. . . . .	70
A.1	Critical action graph of IRON . . . . .	81
A.2	Critical action graph of ENHANCETABLE . . . . .	81
B.1	Mutual information of action-effect pairs in MINECRAFT . . . . .	83



# List of Tables

3.1	Symbols and examples of an action schema. . . . .	18
5.1	Formulas in MINECRAFT environment . . . . .	44
5.2	The parameter setting of genetic programming. The parameter with two values indicates that the settings are different in two phases. . . .	49
6.1	Performance in SWITCH environment after 5M steps. . . . .	62
6.2	Performance in MINECRAFT environment after 5M steps. . . . .	62
6.3	The generalization performance in four tasks. . . . .	66



# List of Symbols

$\mathcal{S}$	state space in MDP problems
$\mathcal{A}$	action space in MDP problems
$\mathcal{T}$	transition function in MDP problems
$\mathcal{R}$	reward function in MDP problems
$\mathcal{Q}$	variable symbol space
$q$	variable symbol
$s_q$	the value refer to the variable symbol $q$
$\mathbb{E}$	effect variable space
$\mathbb{P}$	precondition variable space
$\psi$	critical action
$(a_\psi, \mathbf{P}_\psi, \mathbf{E}_\psi, F_\psi^{\mathbf{P}}, F_\psi^{\mathbf{E}})$	critical action schema
$\mathbf{P}_\psi$	precondition variables of critical action $\psi$
$\mathbf{E}_\psi$	effect variables of critical action $\psi$
$F_\psi^{\mathbf{P}}$	precondition rules of critical action $\psi$
$F_\psi^{\mathbf{E}}$	effect rules of critical action $\psi$



# Chapter 1

## Introduction

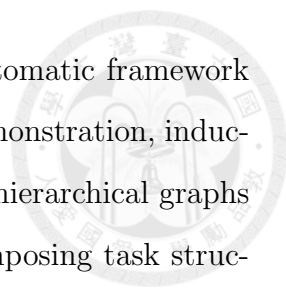
Real-world decision-making problems often exhibit these two key properties: hierarchy and compositionality, which are usual for humans daily. Hierarchical tasks comprise sequential subtasks that must be achieved in a specific order. Before accomplishing each subtask, satisfying the corresponding precondition is essential. Compositional tasks indicate that these subtasks can be reassembled into various tasks. These subtasks possess identical preconditions and effects which can be generalized to unseen tasks. However, the agents often lack the knowledge of task hierarchy and compositionality. Without prior knowledge of task structures and feedback, exploring real-world tasks with these properties remains challenging [23, 50, 2, 42].

To address these issues, the agent's possession of planning capabilities is requisite to effectively tackling the problems, and knowledge representation with compositional structures plays a critical role in planning and reasoning processes [30]. Many approaches have been proposed for hierarchical task structures [32]. Some of these approaches focus on the application of options or high-level policies [44, 23], while others develop curriculum learning methods to guide the agent in a bottom-up task learning process [46]. However, the drawback of these approaches is that they make strong assumptions about pre-defined subtasks, owing to the bottleneck of automatically determining subtasks in diverse environments without domain knowledge.

Due to these assumptions, the adaptability and applicability of these approaches are restricted when applied in real-world scenarios.

In contrast, from a cognitive science perspective, humans can rapidly grasp rules by observing the behaviors of others and their corresponding outcomes [8, 51]. Likewise, intelligent agents exhibit generalization in two aspects. Firstly, task generalization enables agents to solve compositional tasks by transferring skills from previous tasks. We leverage these general skills to solve various tasks with similar subtasks. Secondly, skill generalization enables agents to associate their own skills and adapt to learn related skills. These two-level generalizations allow agents to tackle complex decision-making problems from previous learning. Once the goal is given, humans employ prior knowledge to decompose the tasks into known subtasks and hierarchically accomplish the desired goal. Inspired by the observation, we devise an automatic framework for learning symbolic rules and task structures from expert demonstrations. By leveraging induced prior knowledge, this framework aims to effectively facilitate the agent to achieve the tasks.

The proposed method combines the strengths of reinforcement learning (RL) and classical planning. RL is an inductive learning approach in which an agent interacts with an environment and learns a policy by taking actions and receiving feedback as rewards. Deep reinforcement learning (DRL) has been developed in recent years, integrating RL techniques with deep learning methods. DRL enables agents to effectively solve high-dimensional decision-making problems once deemed intractable [4]. However, DRL poses the drawback of sample efficiency, necessitating substantial experience for training. On the other hand, classical planning is a deductive approach that utilizes symbolic knowledge and logical deduction to determine action plans. Classical planning relies on a pre-defined domain-dependent knowledge base for scheduling, which is impractical in real-world scenarios. Therefore, we propose a method that extracts and utilizes the representation in planning language to facilitate DRL agents in training efficiency. This provides insight into integrating the techniques of high-level abstraction and low-level execution.



The main contribution of our work is that we devise an automatic framework that encompasses the procedure of extracting knowledge from demonstration, inducing planning languages and rules for each subtask, and building hierarchical graphs from goals. This architecture enables task generalization by composing task structure from subtasks to deal with unseen tasks. Additionally, we introduce genetic programming to induce symbolic rules to the framework. Evolutionary methods offer advantages in terms of flexibility and adaptability [6], which promote skill generalization. We claim that it can efficiently adapt to similar rules in few-shot demonstrations once the rules are established.

## Thesis Objective

The proposes of the thesis are listed below:

- Designing a framework to solve hierarchical and compositional decision-making tasks by integrating inductive and deductive learning. The framework utilizes classical planning and DRL and leverages automatically induced rules from demonstrations for planning.
- Introducing genetic programming for symbolic regression, generating symbolic programs to describe the domain knowledge. The method enables the framework to induce flexible symbolic rules from demonstrations.
- Proposing an architecture that captures the knowledge specific to the domain, leading to enhanced generalizability across different tasks. The approach enables task generalization by composing task structures from induced subtasks and skill generalization by adapting to varied symbolic rules.

In this work, an automatic framework is devised to learn symbolic rules and task structures from expert demonstrations. The framework infers abstracted knowledge and constructs the task structure to extract the knowledge from raw data. The

DRL agents follow the navigation from the inferred structures and rules by providing intrinsic rewards. This allows the agents to learn various complex tasks.



## Roadmap

The rest of the thesis is organized as follows:

- **Chapter 2** provides the background and related works of the thesis. Previous studies about learning from demonstrations and subtask inference are introduced in this chapter.
- **Chapter 3** introduces preliminaries, problem definitions, and the concept of critical actions.
- **Chapter 4** elaborates on the proposed framework and approaches, including the workflow and the modules in each step.
- **Chapter 5** provides the descriptions of the test problems to evaluate the capability of hierarchical and compositional task learning.
- **Chapter 6** presents the experiment results and discussion of the proposed framework. The limitations of the work are discussed in this chapter.
- **Chapter 7** gives the conclusions and future works where the analysis and application of critical actions can be further investigated.



## Chapter 2

# Background

This chapter presents the research about hierarchical decision-making problems throughout the domains of classical planning and DRL. The field of classical planning provides valuable insights into preconditions, effects, and their relationship with task structures and transitions. The progress of DRL in recent years also provides a method to efficiently learn the policy. Our proposed framework integrates DRL and planning to improve training efficiency and transfer knowledge by automatically inducing the knowledge.

In addition, drawing inspiration from prior works about state abstraction and subtask inference, we proposed a graph structure representation called critical action graph for representing task structures. By incorporating these ideas, the research endeavors discussed in this chapter contribute to advancing our understanding and addressing challenges associated with hierarchical decision-making in the aforementioned domains.

Sections 2.1, 2.2 and 2.3 provide a comprehensive overview of the core concepts underlying the proposed framework. These concepts encompass DRL, planning, and evolutionary computation. Section 2.4 provided a thorough examination of the related works. This comprehensive exploration establishes a solid foundation for the subsequent discussions and analyses conducted throughout this research.



## 2.1 Deep Reinforcement Learning

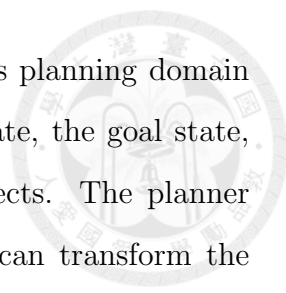
In recent years, RL has witnessed remarkable progress, particularly in DRL. This advancement can be attributed to the rapid growth and capabilities of deep learning, which excels in extracting representations from complex data [4]. The incorporation of deep learning has empowered agents to tackle intricate decision-making problems that were previously considered insurmountable. With DRL, agents can effectively navigate high-dimensional and challenging environments, leveraging deep neural networks' robust representation learning capabilities.

Despite these significant strides, learning hierarchical tasks with sparse rewards remains a persistent challenge for DRL [23]. One of the contributing factors is sample inefficiency. Many real-world problems comprise multiple subtasks, and efficiently learning these hierarchical relationships is crucial for achieving superior performance.

In our framework, DRL serves as an execution module for adaptation in environments. Once abstract knowledge structures are established, intrinsic rewards are provided when the agent executes these skills in the desired order to achieve the tasks. Given the skills and intrinsic rewards, the agent optimizes its policy to complete each subtask properly.

## 2.2 Classical Planning

Classical planning, another subfield of artificial intelligence, aims to develop autonomous strategies for solving planning and scheduling problems [5]. Unlike RL, planning agents use pre-defined symbolic rules to determine the action to reach the goal. Specifically, classical planning excels at finding action sequences to attain desired goals in deterministic and known environments. Nevertheless, in unknown stochastic environments, classical planning agents encounter challenges when exploring these environments due to limitations in the expressiveness of models.



In classical planning, an agent or planning system considers planning domain definition language (PDDL) [14, 40], consisting of the initial state, the goal state, and a set of available actions with their preconditions and effects. The planner uses this information to search for a sequence of actions that can transform the initial state into the desired goal state while satisfying any constraints or conditions specified.

Researchers in classical planning have developed efficient and effective planners that can handle large-scale problems. Once the context of PDDL is defined, the plans can be optimized for different criteria using off-the-shelf planners. However, it requires pre-defined language for reasoning, which relies on predefined languages and structures. This limits their applicability to general real-world problems. In this study, we propose a novel approach that combines classical planning with data-driven techniques, such as DRL and evolutionary computation. This integration allows classical planning to leverage its inherent strengths while effectively addressing the challenges posed by unseen problems.

## 2.3 Evolutionary Computation

Evolutionary computation is a computational approach inspired by the process of natural evolution. Instead of explicitly programming algorithms to search for solutions, evolutionary computation relies on the process of natural selection and evolution to iteratively improve and refine candidate solutions over multiple generations. This technique is widely used to solve complex optimization and black-box search problems in various problems. Due to the mechanism of selection by fitness, evolutionary computation is regarded as a general method for search problems.

In this study, evolutionary computation is regarded as a means to explore and optimize the space of possible knowledge representations and learning algorithms. We employ genetic programming to implement symbolic regression for rule gener-

ation. Symbolic regression differs from deep learning, as it offers flexibility in generating models or programs with domain-specific operators that can adapt to the given data. Genetic programming evolves the programs represented as expression trees based on the data and corresponding outputs.

Our approach uses genetic programming and symbolic regression to generate symbolic rules tailored to the specific problem domain. The process involves the evolution of a diverse set of programs, which enables the exploration of different program structures and compositions to find the most suitable programs to describe the rules. This allows for discovering rules that accurately capture the underlying patterns and relationships within the data.

## 2.4 Related Works

Our framework encompasses the prior works, including learning abstraction from demonstration and subtask inference. Learning abstraction from demonstration provides methods that extract the state and action abstraction from raw demonstrations. We utilize abstract knowledge to infer the task structure and rules. In this section, we introduce these works that focus on these topics.

### 2.4.1 Learning Abstraction from Demonstrations

State abstraction plays a crucial role in enhancing the agent’s reasoning capabilities in high-level planning. By extracting symbolic representations from low-level states, the agent becomes capable of generalizing knowledge across tasks [1]. Several studies have showcased the effectiveness of state abstraction in decision-making problems [1, 16, 15].

One of the methods of learning abstraction is learning from demonstrations. Demonstrations are traces of actions and corresponding states generated by opti-

mal or sub-optimal policies. Agents can learn the policies from demonstrations since demonstrations encompass valuable information regarding task composition and relevant features [8]. Some methods were developed to extract task decomposition and abstraction from demonstrations [17, 9]. This work extends these approaches to introduce inferred knowledge to RL.

## 2.4.2 Subtask Inference

Several approaches that leverage hierarchical and compositional structures of tasks are devised with the various implementation of planning and reasoning. The prior works focus on two subfields: hierarchical task learning and compositional task learning. Hierarchical task learning focuses on dividing tasks into smaller subtasks, while compositional task learning focuses on learning multiple tasks by composing subtasks with different task structures. These two subfields significantly overlap since the compositional subtask is often decomposed from hierarchical tasks. Both of them manipulate subtasks to construct task structures.

### Hierarchical Task Learning

For solving complex problems in RL through task decomposition and scheduling of subtasks at a higher level, a proper hierarchical structure is crucial to represent the task. Various methods have been proposed for constructing hierarchical structure representation [32], including graph [46], automata [13, 47, 22, 50], programs [45], and hierarchical task networks [17, 42]. On the other hand, some approaches utilize the capabilities of deep learning with the use of intrinsic rewards [23, 11].

Despite the success of these methods in building hierarchical models shown in previous work, how to induce the required subtasks had not been addressed. Hence, we develop a method to identify the subtasks by inducing symbolic knowledge and utilizing them as the atom of hierarchical task representation.

## Compositional Task Learning

On the other hand, research works specifically address how to leverage compositional tasks and transfer the knowledge to multiple tasks. The work of these fields seeks to utilize shared information and relationships between tasks to improve overall performance. Some works in this field emphasize the decomposition of skills and the ability to flexibly compose tasks [2, 43, 49, 26, 45]. Some approaches in hierarchical task learning also support compositional task learning [41, 12, 45]. By composing the task structure, generalizability can be realized to solve unseen tasks.



## Chapter 3

# Problem Formulation and Preliminaries

In this chapter, we commence by introducing the preliminaries and problem definitions of the decision-making problems within the context of the Markov decision process (MDP) and PDDL. Subsequently, we delve into the core concepts of our framework, including critical actions and critical action graphs, and highlight their connections to preconditions and effects in classical planning.

### 3.1 Markov Decision Process

The proposed framework aims to solve decision-making problems with hierarchical and compositional properties. Therefore, we focus on sparse-reward goal-directed problems in RL, which can be formulated as MDP problems illustrated in Figure 3.1. MDP can be expressed as a four-tuple  $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$ , where  $\mathcal{S}$  denotes state space,  $\mathcal{A}$  denotes action space,  $\mathcal{T}$  denotes transition function and  $\mathcal{R}$  denotes reward function. To distinguish the states and actions in RL and planning domain, we use *MDP states* and *MDP actions* to denote the states in  $\mathcal{S}$  and the actions in  $\mathcal{A}$ , respectively.

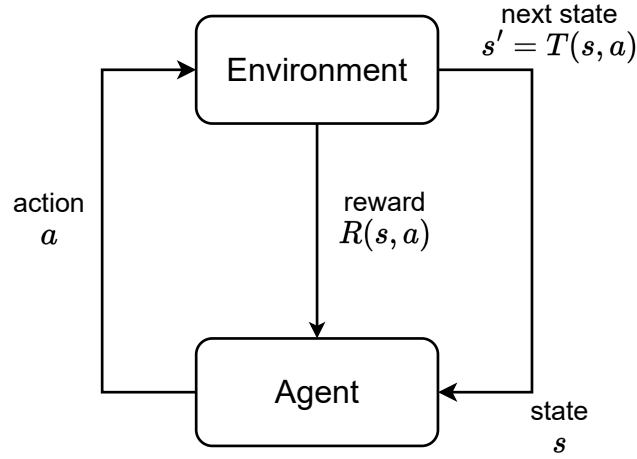


Figure 3.1: The illustration of reinforcement learning in MDP problems. The agent observes the state  $s$  and executes the action  $a$ ; then the environment gives the agent reward  $R(s, a)$  and the next state  $s' = T(s, a)$  based on  $s$  and  $a$ .

Similar to planning problems, the objective of the goal-directed problem is to execute a sequence of actions to achieve a desired goal. The difference is that the RL agent receives rewards in each step, while in the sparse-reward scenario, the agent only receives rewards when achieving tasks.

## 3.2 Planning Domain Definition Language

PDDL is a planning language using first-order logic to express the states and actions in a specific domain. In PDDL, actions are regarded as operators transferring the initial state to the goal state. The preconditions and effects of each action specified the usage of the operator represented as symbolic formulas.

In this work, the problems of building task structures are transferred into planning problems by employing PDDL. To harness the strength of the planning language, formulated domain and problem descriptions in PDDL from MDP problems are required. The essential components of the domain description include the specification of objects, predicates, and actions with preconditions and effects. The

problem description includes an initial state and goal specification. These specifications allow off-the-shelf planners to deduce the optimal action sequence for achieving the desired goal.

Incorporating the representation of PDDL allows us to address complex MDP problems in a structured and reusable manner. It also empowers agents to develop high-level planning capabilities to achieve tasks by clearly and intuitively representing the task structure and goals. In the next section, we will introduce the definition of the state and the action model of MDP problems in the context of PDDL.

### 3.3 MDP Problems in Planning Language

In original MDP problems, each state consists of a list of  $n$  values that describe the features of the worlds. In the planning domain, the features can be represented as variables. Assuming that the meaning of the features from MDP environments is known, we define variable symbol space  $\mathcal{Q}$  as a set containing all  $n$  variable symbols. Each symbol  $q \in \mathcal{Q}$  represents a specific state feature, and each feature has a distinct value represented as  $s_q \in \mathbb{R}$ . In this work, we focus on integer values and denote  $s_q \in \mathbb{Z}$ . That is, an MDP state refers to the list of value  $[s_{q_1}, s_{q_2}, \dots, s_{q_n}]$ , and the state can be transformed into a PDDL state as a conjunction of symbolic formulas

$$\bigwedge_{q \in \mathcal{Q}} (q = s_q). \quad (3.1)$$

For instance, in Figure 3.2, we present a simplified example of the MINECRAFT environment, which is further detailed in Chapter 5. The figure shows a 4×4 grid-world with a wood-picking place, a workbench, and an agent. The agent possesses information about the inventory and its current location, which is represented in

$$\mathcal{Q} = \{\text{wood}, \text{stick}, \text{at\_wood}, \text{at\_workbench}\},$$



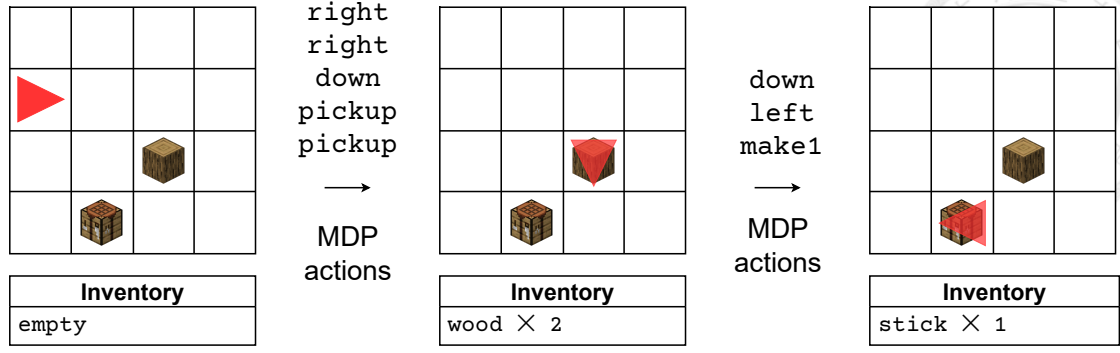


Figure 3.2: An simplified example in MINECRAFT.

where **wood** represents the number of woods in the inventory, **stick** represents the number of sticks in the inventory, **at\_wood** means whether the agent is at the wood-picking place and **at\_workbench** means whether the agent is at the place of workbench. Action space  $\mathcal{A}$  is defined as

$$\mathcal{A} = \{\text{up, down, left, right, pickup, make1}\}.$$

In this task, the agent can pick up a wood by executing **pickup** at the wood-picking place and making a stick by executing **make1**, consuming two woods at the workbench. The objective of the agent is to get a stick. Initially, the agent's inventory is empty and not located at either the wood-picking place or the workbench. Hence, the initial state is represented as

$$(\text{wood} = 0) \wedge (\text{stick} = 0) \wedge (\text{at\_wood} = 0) \wedge (\text{at\_workbench} = 0).$$

The goal specification of the task is  $(\text{stick} = 1)$ . After the agent picks two woods and remains at the wood-picking place, the state can be expressed as

$$(\text{wood} = 2) \wedge (\text{stick} = 0) \wedge (\text{at\_wood} = 1) \wedge (\text{at\_workbench} = 0).$$

Then the agent goes to the workbench and makes a stick. After making a stick and consuming two woods, the state is change to

$$(\text{wood} = 0) \wedge (\text{stick} = 1) \wedge (\text{at\_wood} = 0) \wedge (\text{at\_workbench} = 1).$$

Previous works encode tasks in binary representation to denote the completion or satisfaction of each subtask [21, 42]. However, binary encoding is adverse to generalization due to the growing dimensionality of input, making identifying the relationships between variables impractical. Therefore, we consider numeric variables as more compatible representations. By doing so, our framework can induce symbolic rules to handle more complex tasks that share common domain knowledge. Notably, numeric variables can be handled using first-order logic in the context of PDDL. This enables the agent to present generalizability by manipulating the symbolic rules in task and skill levels.

### 3.4 Critical Action

In this section, we introduce the core concept of the proposed framework: critical actions. A critical action is an action essential in the progress of tasks. In original MDP problems, some actions are used for general purposes, such as movement or changing directions, while others are crucial for the task and must be performed in a specific order. These actions correspond to subtasks in hierarchical tasks. Precisely, certain effects change the state to meet the goal specification. By reasoning from the goal specification, the critical effects necessary to achieve the goal are determined, and the preceding critical effects are the effects that satisfy the precondition of the succeeding one. This leads to a recurrent definition of the effects of the action model in planning. Based on the definition of critical effects, we define the critical action that specifies the critical effects and the required preconditions. Based on the effects, we induce the related MDP action and preconditions from demonstrations to

construct a critical action model. By formulating the critical actions of the problem, we can search for the task structure for variable compositional tasks.

In this research, we employ deterministic critical action models whose assumption is that the symbolic rules are invariant with the tasks. For instance, in MINECRAFT, making a stick always consumes two woods. However, when dealing with real-world scenarios, some configurations may be variant, such as the position of the workbench or the probabilistic movement. In such cases, the deterministic model may not be suitable. However, the issue can be solved by DRL, which can adapt to probabilistic and variant rules through inductive learning. Integration of DRL and classical planning lead to a synergy that efficiently deals with different features of the environment.

### 3.4.1 Effect and Precondition Variable Space

To efficiently induce the rules, we assume that the meaning of the features in a state are known. The properties mark whether the feature depends on hierarchical subtasks and whether agents can change the features. There are two categories to describe the features:

- Precondition variable space  $\mathbb{P}$  indicates the variables which will change over time and can be affected by agents. For instance, `at_wood` is a precondition variable, as `(at_wood = 1)` is the precondition of `(pick_wood)`.
- Effect variable space  $\mathbb{E}$  indicates the variables which directly affect the execution of the subtask and change after executing critical action. These features indicate the progress of the tasks. For instance, `wood` is an effect variable.

For instance, `wood`, `at_workbench` and `at_wood` in MINECRAFT are considered precondition variables since agents can change them and will affect the execution of some subtasks, such as `pickup_wood` or `make_stick`. However, `stick` and `wood` are considered effect variables since the value will change after executing `pickup_wood`

or `make_stick`, while `at_wood` is independent of the progress of the tasks. The variables that belong to neither effect variables nor precondition variables are atemporal. To construct critical actions, we search for the preconditions and effects in the corresponding variable space. The rules of such variables can be learned independently and can be learned by DRL agents.

In the following section, we will introduce the action schema of critical action. We provide the specification of the action schema, including MDP action, variables and rules of precondition and effect. By determining the action schemata of the critical actions, the task structure can be deduced by composing these critical actions.

### 3.4.2 Action Schema

In PDDL, an action schema contains the specification of how an action causes effects and what preconditions are required to execute the action [36]. Let  $\psi$  be a critical action, the action schema of  $\psi$  is defined as a tuple  $(a_\psi, \mathbf{P}_\psi, \mathbf{E}_\psi, F_\psi^{\mathbf{P}}, F_\psi^{\mathbf{E}})$ . The denotations are listed below:

- $a_\psi \in \mathcal{A}$  denotes the MDP action that can be executed if the current state entails the precondition of  $\psi$  in MDP problems. Specifically, if the precondition is satisfied, executing  $a_\psi$  will cause the effects. We differentiate between critical action  $\psi$  and action properties  $a_\psi$  in a critical action executed directly by agents in MDP environments. Critical actions are the time-invariant elements in knowledge systems, whereas  $a_\psi$  are the actions executed in environments.
- $\mathbf{E}_\psi \subseteq \mathbb{E}$  denotes a set of effect variables where the corresponding features change when executing  $a_\psi$ . Detailed symbolic rules are described in  $F_\psi^{\mathbf{E}}$ .
- $F_\psi^{\mathbf{E}}$  denotes effect rules which is a set of formulas indicating what effect that caused by executing  $a_\psi$ .

Table 3.1: Symbols and examples of an action schema.

Name	Symbol	Example
Critical action	$\psi$	<code>make_stick</code>
MDP action	$a_\psi$	<code>make1</code>
Effect variables	$\mathbf{E}_\psi$	<code>{wood, stick}</code>
Effect rules	$F_\psi^{\mathbf{E}}$	<code>{(wood - 2), (stick + 1)}</code>
Precondition variables	$\mathbf{P}_\psi$	<code>{at_workbench, wood}</code>
Precondition rules	$F_\psi^{\mathbf{P}}$	<code>{(at_workbench = 1), (wood ≥ 2)}</code>

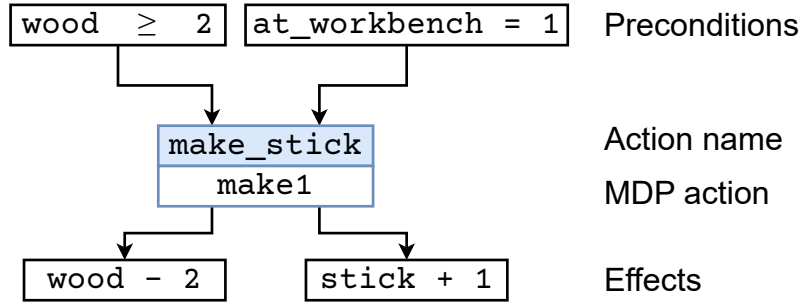


Figure 3.3: The illustration of `make_stick` action schema.

- $\mathbf{P}_\psi \subseteq \mathbb{P}$  denotes a set of precondition variables related to the features that are required to be specific values for executing  $a_\psi$ . Detailed symbolic rules are described in  $F_\psi^{\mathbf{P}}$ .
- $F_\psi^{\mathbf{P}}$  denotes precondition rules which is a set of formulas indicating what situation should be satisfied for executing  $a_\psi$ .

For instance, the critical action `make_stick` is shown in Table 3.1 and Figure 3.3. The effect rules  $F_{\text{make\_stick}}^{\mathbf{E}}$  is `{(wood - 2), (stick + 1)}` where  $\mathbf{E}_{\text{make\_stick}}$  is `{wood, stick}`, and the precondition rules  $F_{\text{make\_stick}}^{\mathbf{P}}$  is `{(wood ≥ 2), (at_workbench = 1)}`, where  $\mathbf{P}_{\text{make\_stick}}$  is `{wood, at_workbench}`. This critical action describes the situation that if the agent executes  $a_{\text{make\_stick}}$  and  $(\text{wood} \geq 2) \wedge (\text{at\_workbench} = 1)$  are satisfied, the effects `(wood - 2)` and `(stick + 1)` will occur.

For convenience in the description in this work, we name the critical action (*e.g.* `make_stick`) according to their consequence, and it is unknown for the framework. The detailed definition will be discussed in Sections 3.4.3 and 3.4.4.

### 3.4.3 Effect Rule

An effect rule is a function  $f_q : \mathbb{R} \rightarrow \mathbb{R}$  which transfers the specific feature value  $s_q$  to another value  $s'_q$  in the transition.  $F_\psi^{\mathbf{E}}$  is a set of effect rules to describe the change of  $s$  related to  $\mathbf{E}_\psi$  which defined as

$$\mathbf{E}_\psi = \{q \mid s'_q = f_q(s) \ \forall f_q \in F_\psi^{\mathbf{E}} \text{ after action } \psi \text{ executed}\}. \quad (3.2)$$

For instance, the effect rules of `make_stick` are `wood - 2` and `stick + 1`, while  $\mathbf{E}_{\text{make\_stick}}$  is  $\{\text{wood}, \text{stick}\}$ , which indicate that executing `make_stick` will acquire one stick and consume two woods. That is, if the current state is represented as

$$\begin{aligned} &(\text{wood} = 4) \wedge (\text{stone} = 3) \wedge (\text{stick} = 0) \wedge (\text{iron} = 0) \\ &\wedge (\text{at\_wood} = 0) \wedge (\text{at\_stone} = 0) \wedge (\text{at\_iron} = 0) \\ &\wedge (\text{at\_workbench} = 1) \wedge (\text{at\_toolshed} = 0), \end{aligned}$$

after executing `make_stick`, the state will be transferred to

$$\begin{aligned} &\underline{(\text{wood} = 2)} \wedge (\text{stone} = 3) \wedge \underline{(\text{stick} = 1)} \wedge (\text{iron} = 0) \\ &\wedge (\text{at\_wood} = 0) \wedge (\text{at\_stone} = 0) \wedge (\text{at\_iron} = 0) \\ &\wedge (\text{at\_workbench} = 1) \wedge (\text{at\_toolshed} = 0). \end{aligned}$$

The underlines in the state highlight the changed variables corresponding to the effects. The effect of the critical action in the context of planning is the conjunction of the effect rules. The rules are the critical effects that are the essence of the critical actions for planning.

### 3.4.4 Precondition Rule

A precondition rule is a logical formula  $f_q : \mathbb{R} \rightarrow \{0, 1\}$  that determines whether the variable  $q$  is satisfied to execute  $a_\psi$ , and  $F_\psi^{\mathbf{P}}$  is a set of precondition rules. Precondition variables  $\mathbf{P}_\psi$  means that all  $q \in \mathbf{P}_\psi$  must met some requirements which is described in  $F_\psi^{\mathbf{P}}$ .  $\mathbf{P}_\psi$  are defined as below:

$$\mathbf{P}_\psi = \{q \mid f_q(s_q) \vee f_q \in F_\psi^{\mathbf{P}} \text{ before action } \psi \text{ executed}\}. \quad (3.3)$$

The precondition of the critical action is a conjunction of  $F_\psi^{\mathbf{P}}$ . For instance, the precondition rules of critical action `make_stick` are `wood`  $\geq 2$ , `at_workbench` = 1 while  $\mathbf{P}_{\text{make\_stick}} = \{\text{wood}, \text{stick}\}$ . This indicate that  $(\text{wood} \geq 2) \wedge (\text{at\_workbench} = 1)$  must be true to execute `make_stick`. That is, it requires at least two woods and stays at the workbench to make a stick. The precondition rules are determined after specifying the critical effects, which describes the necessary conditions for executing critical effects.

## 3.5 Critical Action Graph

In our work, we represent the symbolic knowledge structures as critical action graphs illustrated in Figure 3.4. Given a set of critical actions and a desired goal, a critical action graph is an in-tree structure where the root is the critical action that can achieve the goal directly. The predecessors of a critical action are the required critical actions that should be satisfied before executing a successor critical action. The white blocks show the connection between preconditions and effects of the critical actions; the yellow block denotes the goal of the given task.

Building a critical action graph requires backward search techniques that search for what critical action can satisfy the precondition. For example, in Figure 3.4, to achieve the goal `stick` = 1 with the initial state `stick` = 0, critical action

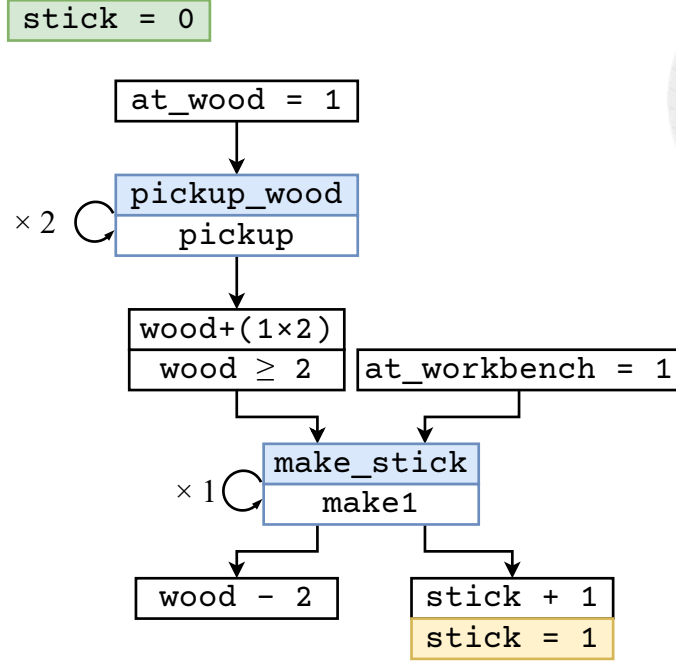


Figure 3.4: The critical action graph for `make_stick`.

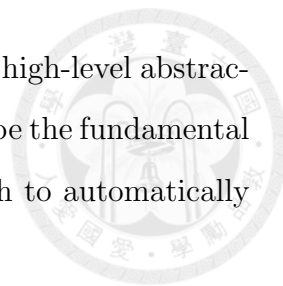
`make_stick` is considered as the root for its effect `stick+1`. Then, to execute `make_stick`, the precondition `wood ≥ 2` and `at_workbench` should be satisfied. Since the `at_workbench` are not in effect variables, we only search for the critical action which can lead to `wood ≥ 2`. Therefore the critical action `pick_wood` is found to become the successor of `make_stick`. The agent needs to execute `pick_wood` two times, which is also marked in the critical action graph. Once the critical action graph is built, agents can leverage the knowledge to solve the task.

Effect variable space  $\mathbb{E}$  are the features that connect each critical action, such as `wood`, and precondition variable space  $\mathbb{P}$  are the features that are not connected with other effects in the critical action graph, such as `at_wood` and `at_workbench` in Figure 3.4. Since only the variables in  $\mathbb{E}$  can be the effects of critical actions, the search space can be reduced by only considering chaining the effect rules and precondition rules whose corresponding variables are in  $\mathbb{E}$  when performing backward chaining.

In summary, we introduce critical actions as the action model in classical planning. By defining critical effects, we establish a connection between the low-level



execution in MDP problems and the action models that represent high-level abstractions in the planning domain. The action schemata serve to describe the fundamental elements of planning. The next section will present the approach to automatically induce critical actions from demonstrations.





## Chapter 4

# Inducing Hierarchical Structure and Symbolic Knowledge

This chapter presents a comprehensive elaboration of the proposed framework. Section 4.1 provides the overview of the proposed frameworks, including the architecture and the procedure. The framework consists of two main modules: induction and training modules. Section 4.2 introduces the induction module that extracts symbolic rules from expert demonstrations and constructs critical action models. Section 4.3 introduces the training module that leverages symbolic knowledge of the models to provide intrinsic rewards during the agent’s training process. These modules are incorporated with the DRL module to learn the low-level execution in MDP environments.

### 4.1 Overview

The architecture of our framework is illustrated in Figure 4.1. The induction module first determines the critical actions from demonstrations offline. The training module deduces the task structure and builds the critical action graph online from the given goal. The critical action graph contains all subtask dependencies which provide

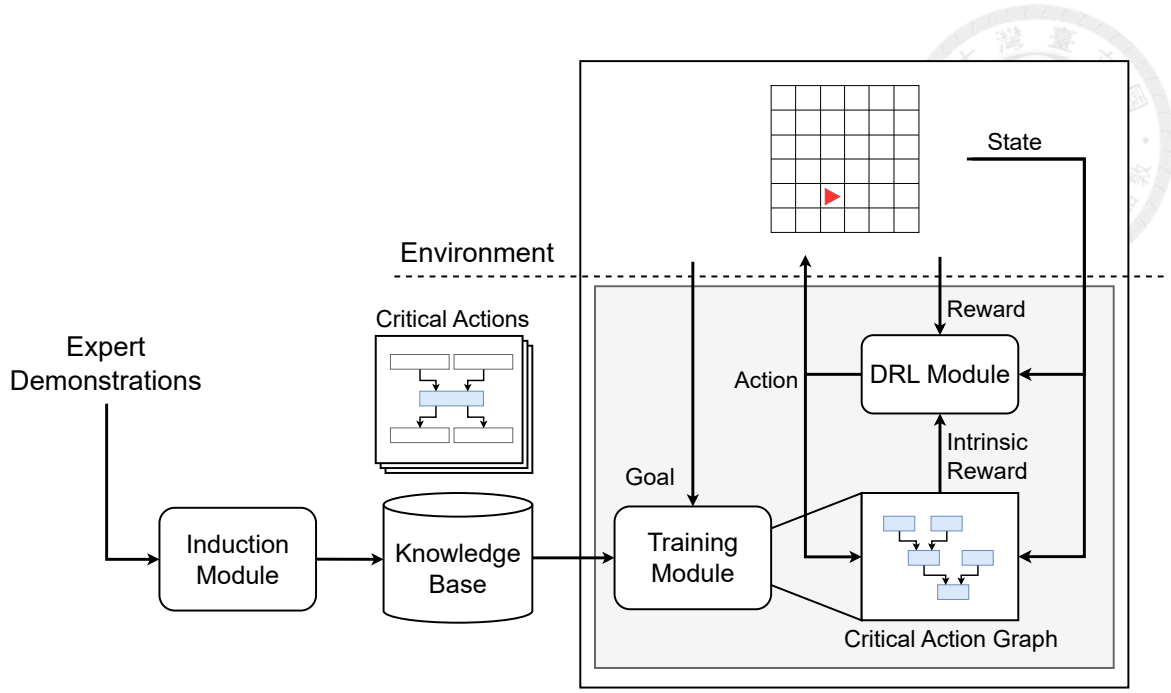


Figure 4.1: The overview of the framework. The induction module infers the critical actions online, and the training module builds critical action graphs and provides intrinsic rewards for agents.

intrinsic rewards for the agent to learn subtasks gradually. While the environment provides extrinsic sparse rewards, intrinsic rewards from our framework incentivize the agent to achieve the desired subtasks with specified effects, augmenting the learning efficiency of the DRL agent. With the hints of intrinsic rewards, the off-the-shelf DRL module learns to adapt to the tasks.

## 4.2 Induction Module

The induction module learns symbolic rules from demonstrations and specifies the critical actions  $\psi$  and their schemata  $(a_\psi, \mathbf{P}_\psi, \mathbf{E}_\psi, F_\psi^\mathbf{P}, F_\psi^\mathbf{E})$ . The procedure of the induction module is illustrated in Figure 4.2. This induction process requires expert demonstrations as input which is explained in Section 4.2.1. In Section 4.2.2, extracting action-effect linkages  $(a, \mathbf{P})$  from demonstrations using mutual information is illustrated. In Section 4.2.3, given  $(a, \mathbf{E})$ , we introduce symbolic regression and

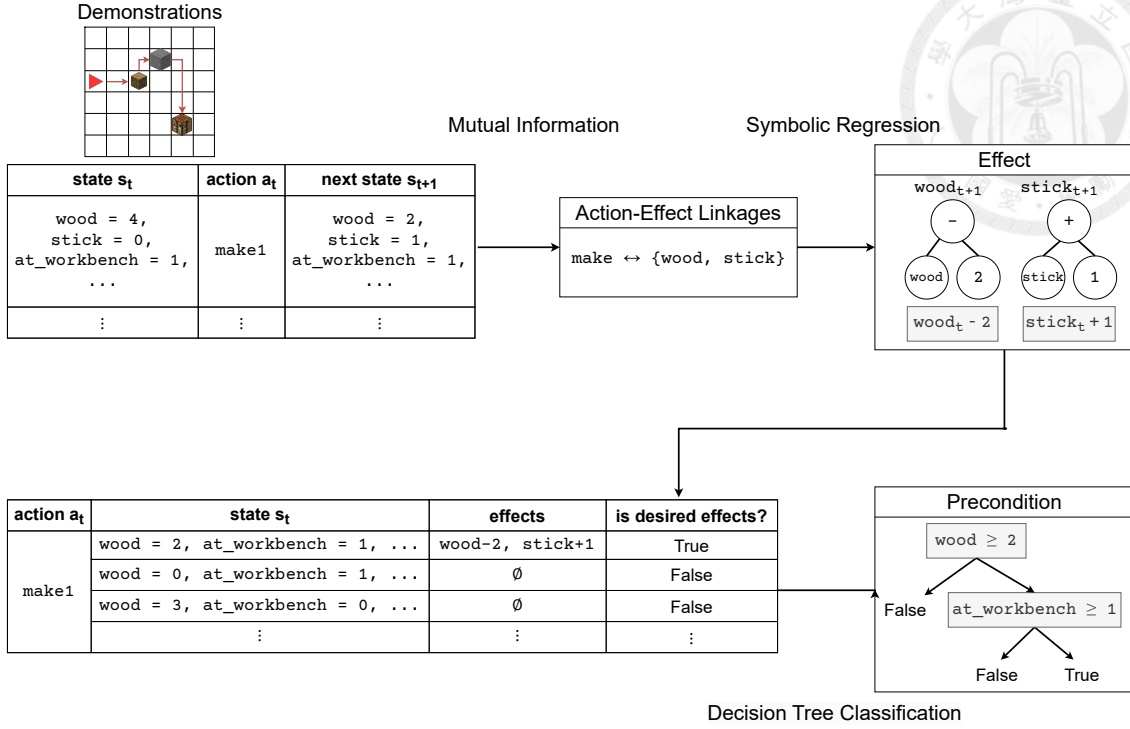


Figure 4.2: The procedure of induction modules.

evolutionary computation to induce effect rules  $F^E$ . In Section 4.2.4, once  $F^E$  is determined, the module leverages the rules to determine the precondition properties  $P, F^P$  for each  $(a, E, F^E)$  using decision trees. After these steps, the components of critical action schemata are all determined. Noticing that we name the critical action  $\psi$  for the convenience of reference, which are not known when inducing action schemata.

### 4.2.1 Collecting Demonstrations

Demonstration is finite alternating sequence of states and actions  $[s^1, a^1, s^2, \dots, s^k, a^k, s^{k+1}]$ , where  $s^t$  denotes the state at time  $t$  and  $a^t$  denotes the action after observing  $s^t$  at time  $t$ . Demonstrations are often generated by human experts, programs, or pre-trained agents. Sampled from an expert's behavior indicates that all demonstrations will successfully reach the goal, which implies the information of task structure and procedure, such as the consequence of the actions and the sequence of the subtasks.

A transition in time  $t$  in demonstrations is defined as  $[s^t, a^t, s^{t+1}]$ , where the effects caused by  $a^t$  refers to the difference between  $s^t$  and  $s^{t+1}$ . Section 4.2.2 introduces the method to extract the action-effect linkages from transitions.

To collect demonstrations, we implement rule-based programs that utilize pre-defined heuristics to make decisions and choose actions. In our frameworks, a sub-optimal policy is acceptable as it helps identify counterexamples for inferring the rules. The framework extracts symbolic knowledge from the underlying information and relation in demonstrations.

## 4.2.2 Extracting Action-Effect Linkages from Demonstrations

The concept of action-effect linkages is based on the outcome assumption that one action only impacts some features of a state. Before searching effect rules, what features are impacted need to be known. Therefore, we will first determine action-effect linkages by calculating mutual information [39] between actions and effects if we observe the effects that often occur after executing  $a$ . The algorithm are shown in Algorithm 1. Let  $\mathcal{E}$  is a set of possible effect variable combinations in the transitions of demonstrations. The mutual information  $M_{(a, \mathbf{E}_{\mathcal{E}})}$  is defined as follows:

$$M_{(a, \mathbf{E}_{\mathcal{E}})} = \sum_{a \in \mathcal{A}} \sum_{\mathbf{E}_{\mathcal{E}} \in \mathcal{E}} P_{\mathcal{AE}}(a, \mathbf{E}_{\mathcal{E}}) \log \frac{P_{\mathcal{AE}}(a, \mathbf{E}_{\mathcal{E}})}{P_{\mathcal{A}}(a) P_{\mathcal{E}}(\mathbf{E}_{\mathcal{E}})}, \quad (4.1)$$

where  $P_{\mathcal{A}}(a)$  is the count of transitions taking action  $a$ ;  $P_{\mathcal{E}}(\mathbf{E}_{\mathcal{E}})$  is the count of transitions that include variables in  $\mathbf{E}_{\mathcal{E}}$ ;  $P_{\mathcal{AE}}(a, \mathbf{E}_{\mathcal{E}})$  is the count of transitions that include changed variables in  $\mathbf{E}_{\mathcal{E}}$  and occur when the agent executes  $a$ . We take the logarithm of mutual information as the metric to avoid incorrect thresholds caused by extremely high mutual information. To determine the linkage, we use two-center clustering, which divides the pairs into two clusters with the threshold of a maximum gap. We select the clusters with higher values as linkages.




---

**Algorithm 1** Extracting Action-Effect Linkages
 

---

**Input:** Demonstrations  $D$ , action set  $\mathcal{A}$ , effect set  $\mathcal{E}$

**Output:** Action-effect pairs with linkages  $L$

```

 $L \leftarrow \emptyset$ 
for  $a$  in  $\mathcal{A}$  do
   $N_a \leftarrow \emptyset$ 
  for  $\mathbf{E}_{\mathcal{E}}$  in  $\mathcal{E}$  do
     $N_a \leftarrow N_a + M_{(a, \mathbf{E}_{\mathcal{E}})}$ 
  end for
 $t \leftarrow \text{Two-Center-Clustering}(M)$ 
  for  $\mathbf{E}_{\mathcal{E}}$  in  $\mathcal{E}$  do
    if  $M_{(a, \mathbf{E}_{\mathcal{E}})} \geq t$  then
       $L \leftarrow L + (a, \mathbf{E}_{\mathcal{E}})$ 
    end if
  end for
end for
  
```

---

For instance, in the task of getting a stick, all combinations of effect variables that occurred in demonstrations are defined as

$$\mathcal{E} = \{\{\text{wood}\}, \{\text{wood}, \text{stick}\}\}. \quad (4.2)$$

Given action space  $\mathcal{A} = \{\text{left}, \text{right}, \text{up}, \text{down}, \text{pickup}, \text{make1}\}$ , we observe linkages between the action `pick_up` and the effect variables `{wood}`, as well as between the action `make1` and the effect variables `{wood, stick}`. By identifying these action-effect pairs, we can narrow down the search space for effect rules to  $\mathcal{E}$ . A transition with  $(a, \mathbf{E}_{\mathcal{E}})$  indicates that the action  $a$  resulted in changes to the variables in  $\mathbf{E}_{\mathcal{E}}$  in the transition. We then effectively induce symbolic rules based on these pairs. Noting that although we have selected the effects with their corresponding actions, we are not certain whether these effects are the critical effects, since the mutual information does not provide the information of dependencies between the effects. The dependencies of critical effects and critical actions are not revealed until the construction of critical action graphs.

### 4.2.3 Determining Effect Symbolic Rules

Once the action-effect pairs with linkage are identified, the induction module proceeds to search for symbolic effect rules denoted as  $F^{\mathbf{E}}$ , caused by action  $a$  for given  $(a, \mathbf{E})$ . The objective is to find each effect rule  $f_q$  in  $F^{\mathbf{E}}$  related to  $(a, \mathbf{E})$ . A rule  $f_q$  is a symbolic program whose input is the state before executing  $a$  and output is the value of variable  $q$  in  $\mathbf{E}$  after executing  $a$ . Let  $s_q^t$  denotes  $s_q$  at time  $t$ . The objective of induction is to find the program that correctly predicts the corresponding value  $s_q^{t+1}$  of variable  $q$  given  $s^t$  in the transitions.

The selection of the symbolic rule model plays a pivotal role in the induction process. The model of rules compatible with the expression of effects facilitates effective inference. For instance, adopting integer numerical expressions as the model for symbolic rules can be effective in integer domains. This can be considered prior knowledge of the problem formulation, which allows our method to acquire rules essential for efficient learning, distinguishing itself from alternative model-free methodologies.

In this work, we employ genetic programming for symbolic regression, with detailed implementation settings provided in Section 5.3.2. Each rule  $f_q$  corresponds to a program representing a variable in  $\mathbf{E}$ . For each run, the goal is to discover the symbolic program that accurately predicts the value of the corresponding variable  $s_q$  in the next state given the current state. This process allows the determination of the rules defined as  $F^{\mathbf{E}}$  corresponding to the effect variables in  $\mathbf{E}$ . The illustrations are shown in Figure 4.3. For instance, to induce the symbolic effect rules of `pickup_wood`, we require the effect variables  $\mathbf{E}_{\text{pickup\_wood}} = \{\text{wood}\}$ . This method allows the induction module to determine the rules for each effect variable in  $\mathbf{E}$ , denoted as  $F^{\mathbf{E}}$ . In the next section, we introduce the method for inducing precondition variables and rules  $\mathbf{P}$ ,  $F^{\mathbf{P}}$  given tuple  $(a_\psi, \mathbf{E}, F^{\mathbf{E}})$ .

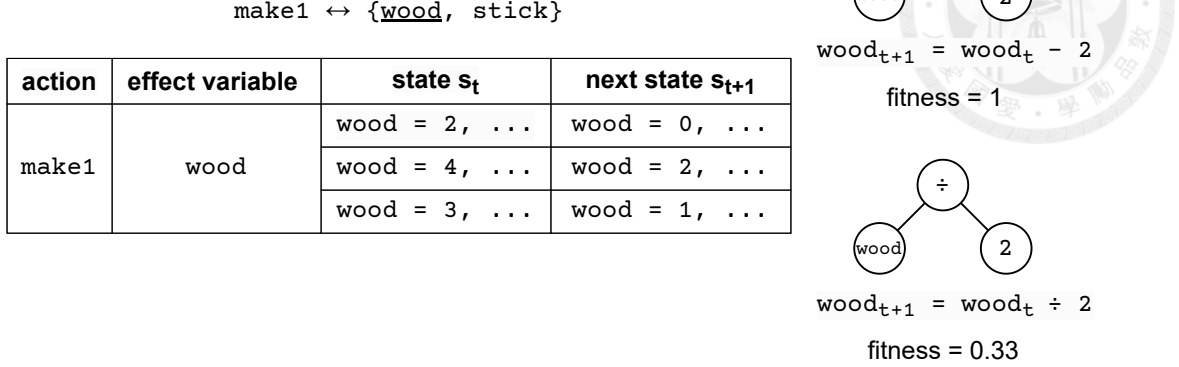


Figure 4.3: The illustration of symbolic regression using genetic programming.

#### 4.2.4 Determining Precondition

After  $F^E$  are found, precondition variables  $\mathbf{P}$  and precondition rules  $F^P$  of critical action are identified by classifying whether the transition is consistent with  $F^E$ . The process involves minimal consistent determination (MCD) and the decision tree method. Given  $a$  and  $F^E$ , the approach should find the formula from transitions  $[s^t, a^t, s^{t+1}]$  in demonstrations. The formula can decide what preconditions should be satisfied in  $s^t$ , leading to desired effects after  $a^t$ .

Assumed that  $s_q^{t+1}$  is the value of variable  $x$  in  $s^t$ , and  $s_q^{t+1}$  is the value of  $q$  after transition, a consistent determination  $x \succ q$  means that if some transitions share the variable  $x$  that have identical values in  $s_x^t$ , then  $s_q^{t+1}$  must be consistent which also have identical values. MCD is the smallest subset of relevant variables that is necessary to predict the specific outcome [36]. For instance, making a stick necessitates the current position at a workbench and at least two woods in MINECRAFT environment. Hence, the MCD of the precondition for critical action **make\_stick** is  $\{\text{wood}, \text{at\_workbench}\}$ . The algorithm of consistent determination and MCD are depicted in Algorithm 2 and 3.

MCD yields a more concise hypothesis to find the decision formula. Combining with MCD, a decision tree algorithm aims to learn the classification of whether the






---

**Algorithm 2** Consistent-Determination

---

**Input:** a subset of features  $F$ ; a set of samples  $S$   
**Output:** Truth Value: if  $F$  is consistent determination of  $S$   
 H: empty hash table  
**for**  $s$  in  $S$  **do**  
    $v \leftarrow$  the values of  $s$  for  $F$   
    $c \leftarrow$  the class of  $s$   
   **if** exist  $(v', c')$  in  $H$  such that  $v = v'$  but  $c \neq c'$  **then**  
     **return** False  
   **end if**  
   insert  $(v, c)$  to  $H$   
**end for**

---



---

**Algorithm 3** Minimal-Consistent-Determination

---

**Input:** a set of features  $A$  with size  $n$ ; a set of samples  $S$   
**Output:** minimal consistent determination  
**for**  $i = 0$  to  $n$  **do**  
   **for** each  $A_i$ : subset of  $A$  with size  $i$  **do**  
     **if** Consistent-Determination( $A_i, S$ ) **then**  
       **return**  $A_i$   
     **end if**  
   **end for**  
**end for**

---

transition is consistent with the rules in  $F^E$  in demonstrations. This integration improves the learning performance of original decision tree algorithms [36].

The proposed framework uses classification and regression tree (CART) [27] to build decision trees. CART is a supervised learning algorithm that generates binary trees by recursive partitioning, where each internal node represents a decision based on a specific variable, and each leaf node represents a prediction.

Let data partitioned at the internal node  $m$  denoted as  $D_m$  with  $n_m$  samples. The algorithm aims to find a decision with a variable  $q$  and a threshold  $t$  to partition  $D_m$  into two subsets  $D_m^0$  and  $D_m^1$  with  $n_m^0$  and  $n_m^1$  samples. The entropy of  $D_m^i$  is defined as below:

$$H(D_m^i) = - \sum_{D_m^i} \frac{D_k^i}{D_m^i} \log\left(\frac{D_k^i}{D_m^i}\right), \quad (4.3)$$

where  $D_q$  represents the data that is correctly classified in  $D_m^i$ .

The loss function of the partition is defined as follows:

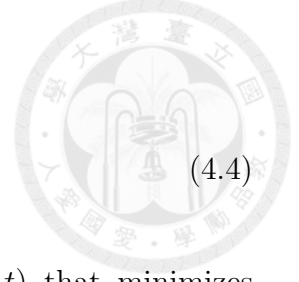
$$G(D_m, q, t) = \frac{n_m^0}{n_m} H(D_m^0) + \frac{n_m^1}{n_m} H(D_m^1). \quad (4.4)$$

In each partition, the algorithm's objective is to find the  $(q, t)$  that minimizes  $G(D_m, q, t)$  at node  $m$ . This process is repeated recursively until a stopping criterion is met.

In the given transition  $[s^t, a^t, s^{t+1}]$  with  $(a, \mathbf{E})$  in demonstrations, the current states  $s^t$  are taken as inputs to a decision tree, and the outcome of the decision tree is a true value that whether  $s^{t+1}$  consistent with the rules in  $F^{\mathbf{E}}$ . After generating decision tree by CART, the model of this decision tree is then transferred into a conjunction of rules by logical simplification and set as the precondition rules  $F^{\mathbf{P}}$ , while  $\mathbf{P}$  is the set of variables mentioned in  $F^{\mathbf{P}}$ . Considering the precondition is the conjunction of the precondition rules while the formula of the decision tree may involve disjunction, the decision tree model is transferred into the disjunctive normal form. Each clause in the disjunctive normal form is considered the precondition for different critical actions.

#### 4.2.5 Summary

After the aforementioned procedure, all action schemata  $(a, \mathbf{P}, \mathbf{E}, F^{\mathbf{P}}, F^{\mathbf{E}})$  is determined. The rules in  $F^{\mathbf{P}}$  and  $F^{\mathbf{E}}$  can be simplified by symbolic deduction. These rules identify whether an action is critical when training an agent. The workflow of our framework is depicted in Figure 4.4. In the next section, we delve into the training module, which involves constructing critical action graphs and providing intrinsic rewards.



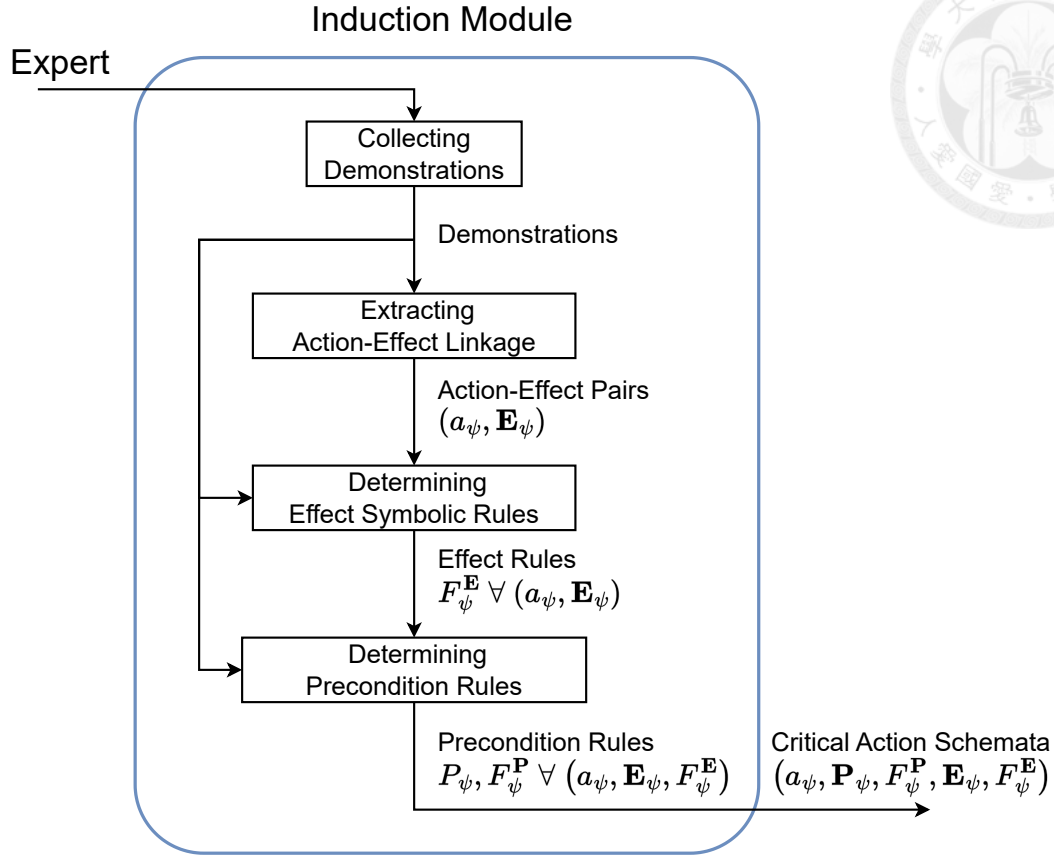
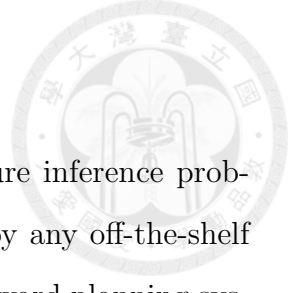


Figure 4.4: The workflow of the induction module with input and output of each method.

### 4.3 Training Module

After the induction process, the critical action schemata serve as the components of a knowledge model that guides the agent in the training stage. The dependencies among the critical actions can be deduced through symbolic computation. During the training stage, the training module derives critical action graphs as the task structures. After establishing the structure, intrinsic rewards are assigned based on whether the execution of the action satisfies the induced action schema in the graph. If the agent successfully performs an action that meets the specified preconditions and effects, it receives an additional intrinsic reward. Further details are discussed in Sections 4.3.1, 4.3.2 and 4.3.3.



### 4.3.1 Inferring Critical Action Graph

Once the critical actions schemata are defined, the task structure inference problem can be transferred into a planning problem and be solved by any off-the-shelf planners such as fast forward planning system [20] and fast downward planning system [18]. Critical action graphs are composed during the training stage to handle various tasks according to the goal that the agent is asked to achieve.

Given a goal  $G$  (*e.g.* `stick = 1`) and an initial state (*e.g.* `stick = 0`), the proposed framework deduces the required critical actions of  $G$  by backward chaining and expresses the dependencies in a tree structure. The illustration of the critical action graph is shown in Figure 3.4. In symbolic computation, the effects of a critical action are regarded as operators, and the precondition of a critical action in the critical action graph should be satisfied. Starting from the goal, the module search for the critical action to find the desired effect for unconnected precondition. Maximum operation steps are set to terminate the search. Once the critical action is found, the critical action will consider as the predecessor of previous critical actions. The number of required execution are also recorded.

### 4.3.2 Deep Reinforcement Learning Agent

In the training stage, we aim to train a DRL agent that can learn the subtask by leveraging the feature-extracting power of neural networks. The induction module only specifies the coarse-grained critical action to express temporal order. Therefore, the framework utilizes DRL to complete the fine-grained decision-making tasks.

RL is a reward-directed approach that can be used to solve MDP problems. The goal of RL agents is to sequentially find environment actions that can receive maximum expected accumulative rewards from the environment. Specifically, DRL agents utilize deep learning the approximate the optimal policy with neuron networks. DRL uses the policy gradient method to update the policy. In the proposed

method, we use advantage action-critic (A2C) [28, 10] as the DRL agent, which detail introduction is in Section 5.3.1.

The problem of sparse reward is that the environment only provides information when the task is achieved. Therefore, merely DRL agents are not efficient in solving the tasks. In next section, we introduce the mechanism of intrinsic reward and defined the intrinsic reward based on critical action graph.

### 4.3.3 Intrinsic Rewards

During the training stage, we train an agent with intrinsic rewards generated from the critical action graph. The modified rewards function is illustrated as follows:

$$R_{int}(s) = \begin{cases} +1 & \text{if execute a critical action} \\ 0 & \text{otherwise.} \end{cases} \quad (4.5)$$

That is, if the agent succeeds in executing the critical effects, it will receive an intrinsic reward. However, the number of execution is known and there is a limit on the number of times the agent can receive this intrinsic reward. This prevents the agent from solely focusing on obtaining intrinsic rewards. This method prevent the agent from solely focusing on obtaining intrinsic reward. Given the original reward function  $R_{ext}$  as extrinsic rewards, the overall reward of the MDP problem is

$$R(s) = R_{ext}(s) + R_{int}(s). \quad (4.6)$$

When the preconditions of a critical action  $\psi$  are satisfied, the agent is expected to execute the corresponding action  $a_\psi$ . If the agent successfully achieves the desired effects, it receives an intrinsic reward. This approach is akin to curriculum learning, where simpler tasks are presented to the agent initially to facilitate learning more complex tasks gradually. The intrinsic reward serves as an incentive for the agent to

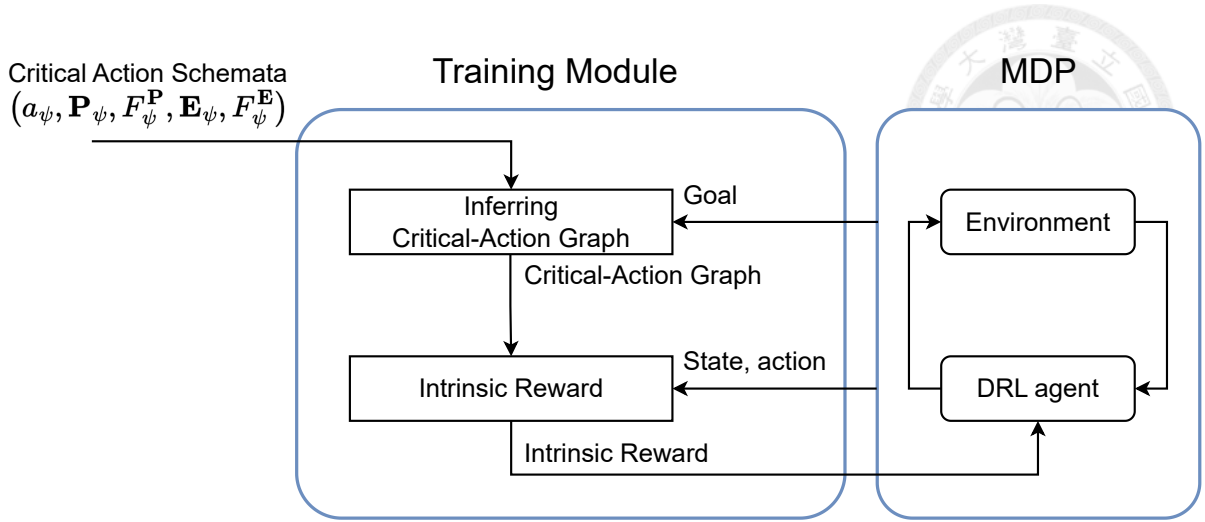


Figure 4.5: The workflow of the training module with input and output of each method.

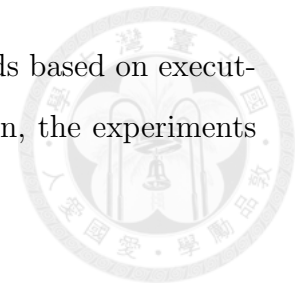
select actions that lead to the desired effects. Conversely, if the agent takes an action that leads to undesired effects, such as violating effect rules, it will receive a penalty. However, it is important to note that our model only specifies positive critical actions and does not explicitly identify actions that may have negative consequences. Therefore, the implementation of a penalty depends on the specific domain.

We serve intrinsic rewards as an augmentation, which allows for applying the methods on off-the-shelf DRL agents, enabling flexibility to enhance the DRL model. Compared to the original DRL agents, our framework’s agents require less time to master jobs with sparse rewards. Particularly in challenging situations, agents with our framework can do tasks unachievable for the original agents.

#### 4.3.4 Summary

In this section, we introduce the procedure of the training module and the components of the module. The workflow of the training module is shown in Figure 4.5. The critical action schemata are considered as subtasks and the critical action graph expresses the dependencies of the subtasks. Backward chaining is employed to determine the dependencies among critical actions. The formation of critical action

graphs as task structures is then used to provide intrinsic rewards based on executing actions satisfying induced action schema. In the next section, the experiments to demonstrate the capability of our framework are shown.





## Chapter 5

# Test Environments and Experiment Settings

In this chapter, we introduce two MDP environments SWITCH and MINECRAFT for evaluation. The first environment SWITCH is focused on assessing the ability to sequentially achieve subtasks in hierarchical tasks. The second environment MINECRAFT is designed to evaluate the ability to construct various task structures with multiple subtasks for compositional tasks. Both environments are set in a  $8 \times 8$  grid. Agents can move in the grid with action `up`, `down`, `left`, `right` and interact with the object on the grid according to the configuration of the environment. For each environment, several tasks are designed for testing effectiveness and generalizability. Detail descriptions are elaborated in Sections 5.1 and 5.2. In addition, in Section 5.3, the parameters and settings of the proposed framework are specified. In Section 5.4, we introduce the algorithms we selected for comparison and their implementation details.



## 5.1 Switch

The environment SWITCH is designed to evaluate the ability to solve hierarchical tasks. In SWITCH, several switches are placed on the grid. The objective of the agent in SWITCH is to sequentially turn on switches in a pre-determined order.

We define the state variables as

$$Q = \{x, y, \text{at\_switch}, \text{next\_switch}, \text{goal\_switch}\},$$

where  $x$  and  $y$  indicate the position of the agent on the grid. `at_switch` indicates the switch the agent stays at. If the agent does not stay at any switch, this variable is set to zero. `next_switch` indicates which switch should be activated in the following actions. `goal_switch` denotes the last switch and also implies how many switches should be turned on. The action space  $\mathcal{A}$  contains five actions: `{left, right, up, down, toggle}`. `toggle` enables the agent to activate or deactivate a switch.

The switches have three states:

- *available*: When the agent executes the action `toggle` on the switch, the switch is activated to *on*.
- *on*: The switch is currently activated. If the agent executes the action `toggle` at the switch, it will be deactivated and turned to *available*.
- *off*: The agent can not change the status of the switch.

When a switch is on, it activates the next switch in the pre-defined order and turns the switch from *off* to *available*. However, if a previously activated switch is turned to *available* (not *on*), all subsequent switches will also be deactivated. This makes it challenging for RL agents to solve the task through random walks or exploration alone.

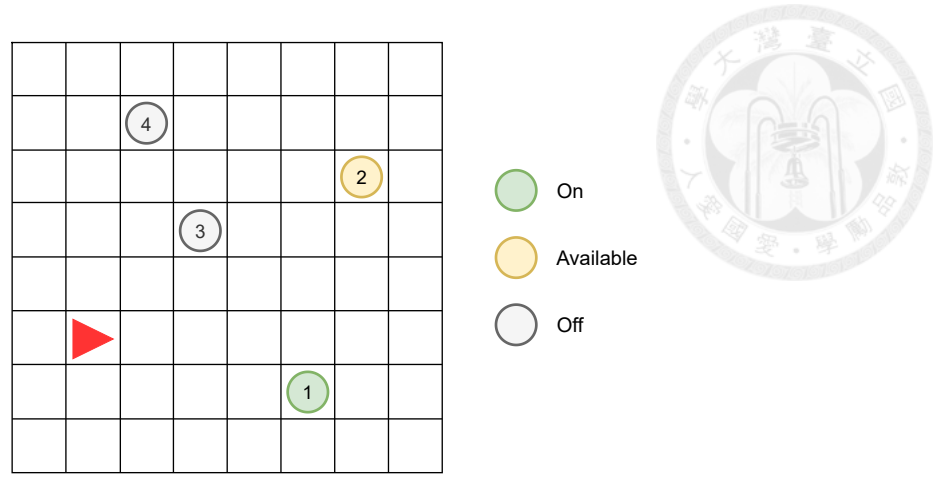


Figure 5.1: Visualization of SWITCH environment. The goal of the agent is to turn on all switches in a specific order. The figure shows the state of 4-SWITCHES-INCREMENTAL. Green circles indicate *on*, yellow switches indicate *available*, and gray switches indicate *off*.

An example is shown in Figure 5.1. Assumed there are  $n$  switches labeled from 1 to  $n$  located in SWITCH, the rule of order is to turn on switches incrementally, and the `goal_switch = 4`. That is, the agent needs to activate switches 1, 2, 3, and 4, respectively. At first, only switch 1 is available, and others are off. The agent is tasked with activating switches in sequential order. After we operate switch 1, switch 1 is set to *on*, and switch 2 is set to *available*. However, if any switch  $i$  in *on* state is deactivated, for all switch  $j$  where  $j \geq i$  are set to *off*, and switch  $i$  are set to *available*.

For instance, when the agent stays at (1, 2) where switch 2 is at and going to turn on switch 2, the state is represented as

$$(x = 1) \wedge (y = 2) \wedge (\text{at\_switch} = 2) \wedge (\text{next\_switch} = 2).$$

After the agent turns on switch 2, the next switch to turn on becomes switch 3, and the state will transfer to:

$$(x = 1) \wedge (y = 2) \wedge (\text{at\_switch} = 2) \wedge (\text{next\_switch} = 3)$$

The state means that switch 2 is on and switch 3 is available. Now the agent should go to switch 3 and turn on the switch. If the agent goes to (2, 2) where there is no switch, the state will transfer to

$$(x = 2) \wedge (y = 2) \wedge (\text{at\_switch} = 0) \wedge (\text{next\_switch} = 3).$$

In addition, we define the difficulty of the tasks on the number of switches. Due to the possibility of deactivation, when the number of switches increases, it becomes intractable for RL agents to achieve the tasks only by exploration. Since the critical actions of the environment remain the same, our framework is capable of generalizing the critical actions from an easy task to a difficult one. To evaluate the capability of the proposed framework, we design several situations, including the number of switches, sequential order, and distractors. In the following sections, the settings of the tasks are listed and discussed.

### 5.1.1 Number of Switches

We use the number of switches to evaluate the performance on different difficulties of tasks. When the number of switches increases, the tasks become more difficult as it has more chance to turn off the switch. We show that our framework enables DRL agents to learn difficult tasks, and the critical action can be generalized to different numbers of switches.

### 5.1.2 Order

To analyze the capability of skill generalization, We define two types of orders including INCREMENTAL and ODD. INCREMENTAL indicates that the switches should be turned on in incremental order. That is, if four switches need to be turned on, we set `goal_switch = 4` and the activate order is 1, 2, 3, and 4. ODD indicates that the switches should be turned on in odd order where `goal_switch = 7` and the

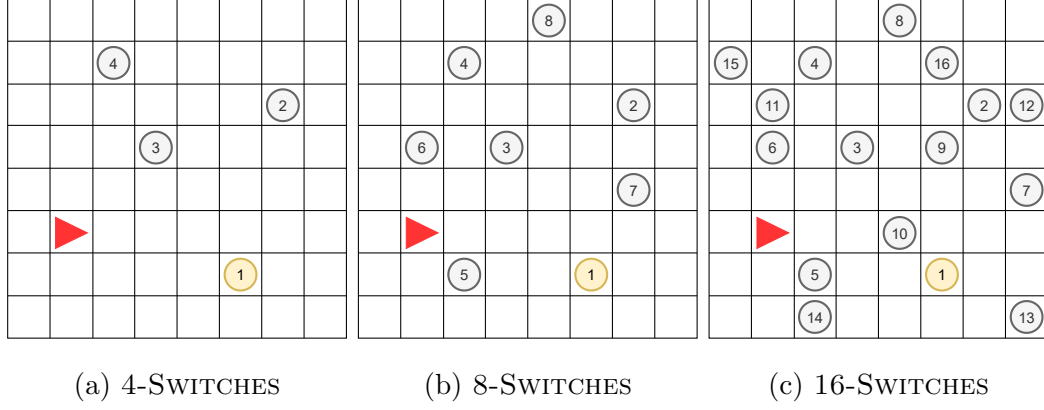


Figure 5.2: Visualization of SWITCH environment with 4, 8 and 16 switches.

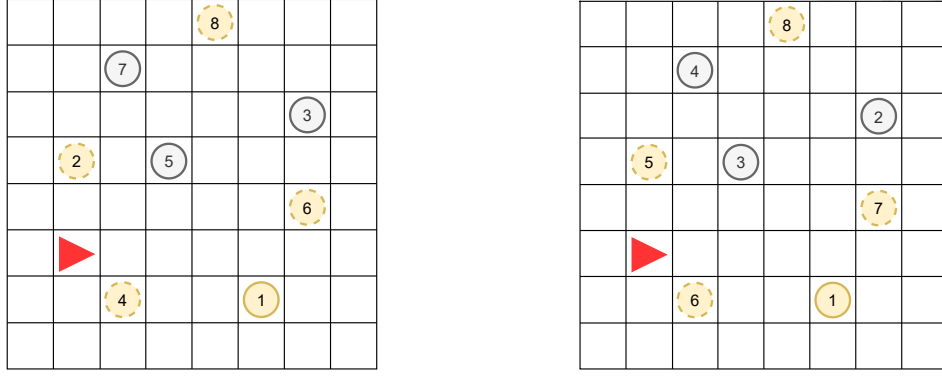
activate order is 1, 3, 5, and 7 if there are four switches. We will show that given the critical actions from the demonstrations in one type of order, the framework can adapt to new rules from few-shot demonstrations in another type of order.

### 5.1.3 Distractors

In the distractor setting, available switches are added to the environment. The agent can turn on and off the distractor switches, but the action does not help to achieve the tasks. In  $n$ -switch incremental-order tasks, the switches labeled  $n + 1$  to  $2n$  are set as distractors. In  $n$ -switch incremental-order tasks, the switches labeled  $2, 4, \dots, 2n$  are set as distractors. This setting evaluates whether the agent acquires the ability to select correct switches and neglects the incorrect ones.

### 5.1.4 Four Rooms

In the four-room setting, two lines of walls divide the gridworld into four rooms according to the four-room configuration in *Minigrid*. Every two rooms are interconnected by a gap in the walls. In this scenario, the agent is required to navigate through the rooms considering the walls to activate the switches. This setting evaluates the efficacy of DRL involving navigating obstacles at low-level execution.



(a) 4-SWITCHES-4-DISTRACTORS-INCREMENTAL (b) 4-SWITCHES-4-DISTRACTORS-ODD

Figure 5.3: Visualization of SWITCH environment with distractors in different order. The dotted stroke circles represent distractors which is initially available.

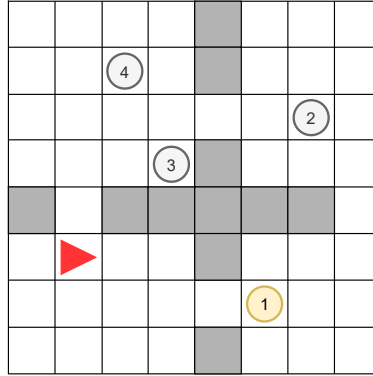


Figure 5.4: Visualization of SWITCH environment with four rooms. The gray squares represent walls which can not move across.

## 5.2 Minecraft

MINECRAFT is inspired by the computer game Minecraft and is similar to the environment in previous works [41, 2, 45, 7]. The agent can pick up the primary materials on the map and make different tools in specific places consuming the materials. The goal of each task is to acquire the desired materials or tools. The state variables include  $\{x, y, \text{at\_place}, \text{inventory}\}$ , where  $\text{at\_place}$  denotes whether the agent is at the  $\text{place}$ , and  $\text{inventory}$  denotes the number of materials or tools the agent holds. For instance,  $\text{at\_workbench} = 1$  means the agent is at the workbench, and  $\text{wood} = 3$  means the agent has three wood.

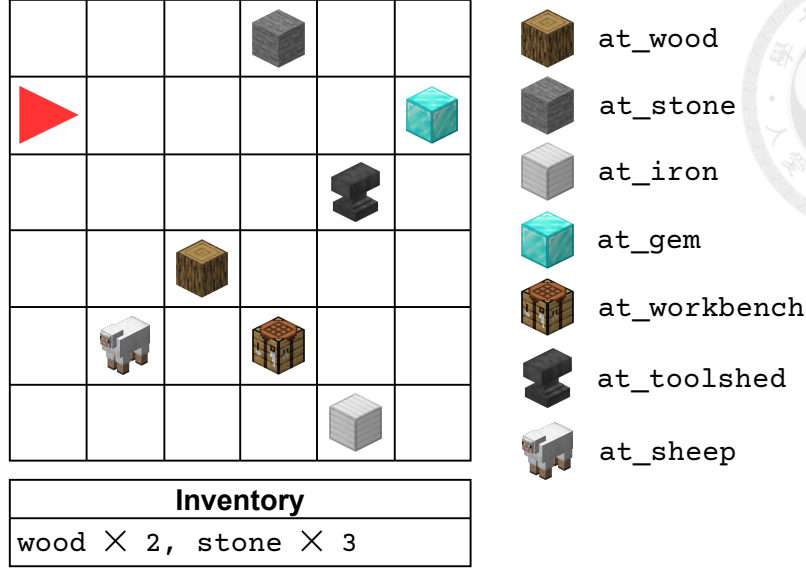


Figure 5.5: Visualization of MINECRAFT environment.

In our experiments, thirteen types of items are designed in the inventory: `wood`, `stone`, `stick`, `iron`, `gen`, `stone_pickaxe`, `iron_pickaxe`, `wool`, `paper`, `scissors`, `bed`, `jukebox`, `enhance_table`. There are seven places on the grid world: `at_wood`, `at_stone`, `at_iron`, `at_gem`, `at_sheep`, `at_workbench`, `at_toolshed`. The state variables  $Q$  are defined as

$$Q = \{\text{wood, stone, stick, iron, gen, stone\_pickaxe, iron\_pickaxe, wool, paper, scissors, bed, jukebox, enhance\_table, at\_wood, at\_stone, at\_iron, at\_gem, at\_sheep, at\_workbench, at\_toolshed}\}.$$

The action space  $\mathcal{A}$  contains eight actions: `{left, right, up, down, make1, make2, make3, make4}`. The agent crafts different items when executing different `make` actions (`make1`, `make2`, `make3`, `make4`) and at different places (`workbench` or `toolshed`). The formulas of the items are listed in Table 5.1, and the dependency of the subtasks is illustrated in Figure 5.6. The agent needs to get the materials to create desired items. We test two single tasks with different difficulties `IRON` and `ENHANCE_TABLE`, and a multiple task `MULTIPLE` that sample the goal at random.



Table 5.1: Formulas in MINECRAFT environment

Inventory	Action	Preconditions	Effects
wood	pickup	at_wood = 1	wood + 1
stone	pickup	at_stone = 1	stone + 1
iron	pickup	at_iron = 1	stone_axe $\geq$ 1 iron + 1
gem	pickup	at_gem = 1	iron_axe $\geq$ 1 gem + 1
wool	pickup	at_sheep = 1	scissors $\geq$ 1 wool + 1
stick	make1	at_workbench = 1	stick + 1      wood - 1
stone pickaxe	make1	at_toolshed = 1	stone pickaxe + 1    stone - 3    stick - 2
iron pickaxe	make2	at_toolshed = 1	iron pickaxe + 1    iron - 3    stick - 2
scissors	make2	at_workbench = 1	scissors + 1      iron - 2
paper	make3	at_workbench = 1	scissors $\geq$ 1 paper + 1      wood - 1
bed	make3	at_toolshed = 1	bed + 1      wood - 3    wool - 3
jukebox	make4	at_workbench = 1	jukebox + 1      wood - 3    gem - 1
enhance table	make4	at_toolshed = 1	enhance table + 1    stone - 3    paper - 2    gem - 1

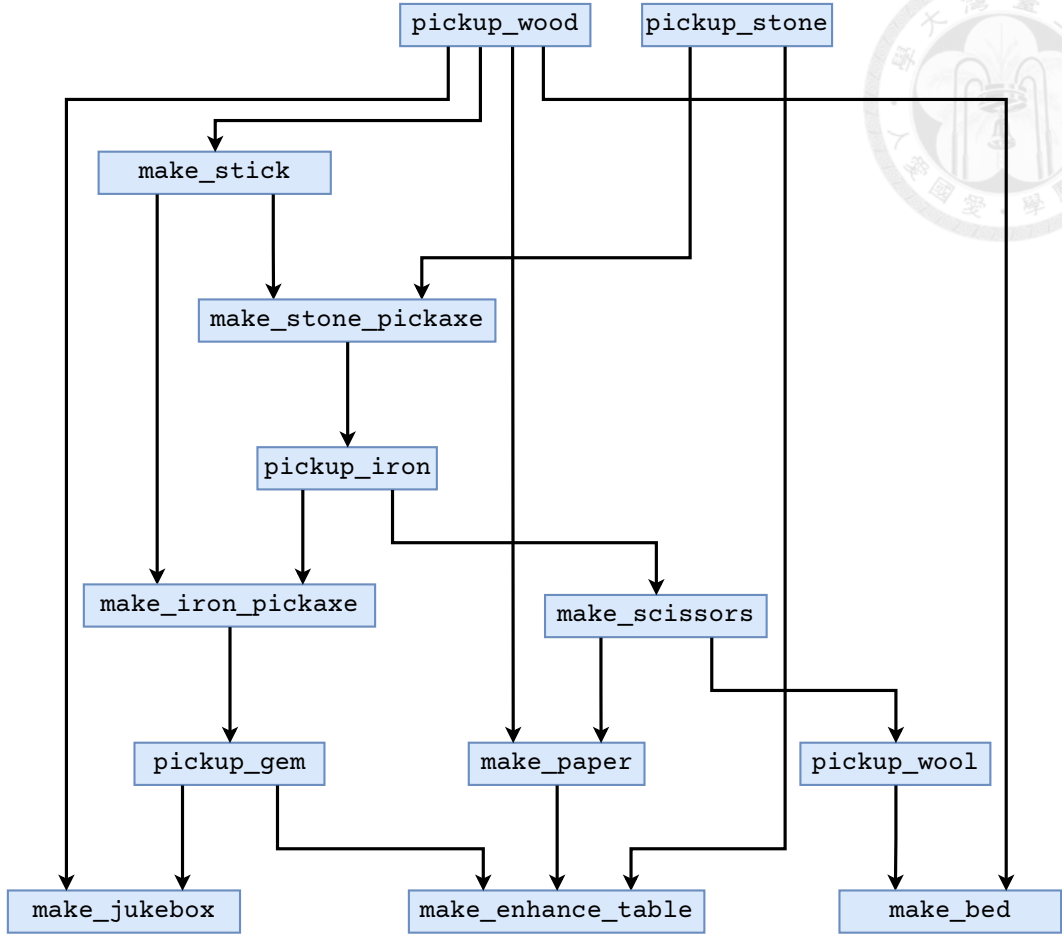


Figure 5.6: Dependency of subtasks in MINECRAFT. We select IRON and ENHANCE TABLE as test problems, and the detail illustration about effects and the preconditions of the problems are shown in Appendix A.

### 5.3 Implementation Detail

In this section, we elaborate on the implementation detail of the proposed framework that was used in the experiments. We implement several modules using off-the-shelf packages and approaches, including the agents with A2C, and genetic programming as symbolic regression with *gplearn*.



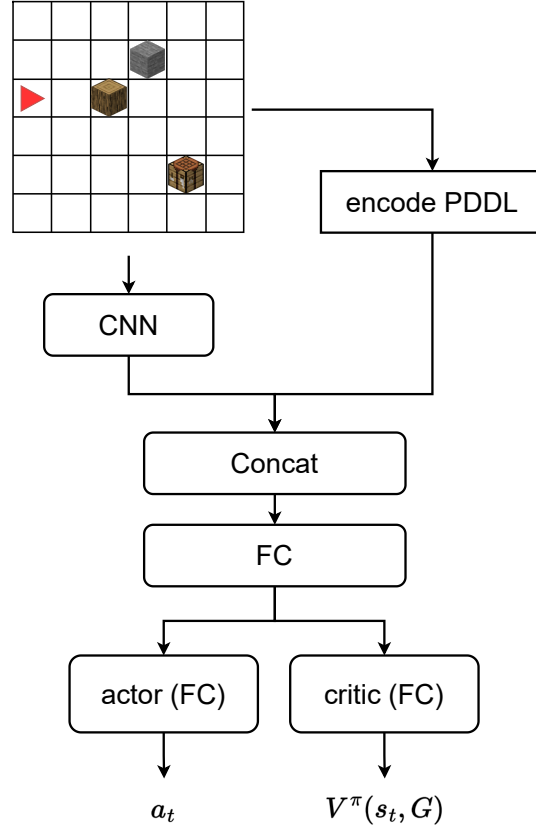


Figure 5.7: The architecture of DRL agents incorporates A2C methods. The actor generates actions, and the critic evaluates the actor's output.

### 5.3.1 Advantage Actor-Critic

A2C are used for the DRL module in our framework. The actor-critic method is a temporal difference method where an actor module is served as a policy, and a critic module is regarded as a value function. A2C implements the advantage function representing the advantage of taking a particular action in a given state compared to the average expected reward. In each step, the agent's policy  $\pi$  with parameter  $\theta$  executes an action according to the state from the environment and consequently receives rewards. After termination, the probability of trajectories  $\tau$  occurred are expressed as

$$p_\theta(\tau) = p(s^1) \prod_{t=1}^T p_\theta(s^t | s^{t-1}, a_t) p(a^t | s^t). \quad (5.1)$$

Given the trajectories  $\tau$ , the policy gradient of expected rewards  $\bar{R}_\pi$  derived from  $\tau$  are provided to update the neuron network which defined as

$$\nabla \bar{R}_\theta(\tau) = \sum_{n=1}^N \sum_{t=1}^{T_n} (r^t + V^\pi(s^{t+1}, G) - V^\pi(s^t, G)) \nabla \log p_\theta(a^t | s^t), \quad (5.2)$$

where  $V^\pi(s^t, G)$  denotes the value function learned by critics with parameters  $\pi$ , which indicate the expected accumulated rewards from  $t$  to end of the episode, and  $r^t + V^\pi(s^{t+1}, G) - V^\pi(s^t, G)$  denotes advantage function which implies the difference between rewards  $r_t$  and the reward estimator  $V^\pi(s^t, G) - V^\pi(s^{t+1}, G)$ . The policy update rule for the actor is

$$\theta' \leftarrow \theta + \alpha \nabla \bar{R}_\theta(\tau), \quad (5.3)$$

where  $\alpha$  denotes the learning rate. The objective of A2C is to minimize cross-entropy of  $\nabla \bar{R}_\pi$ .

In this work, *torch-ac* [25] is used for the implementation of the DRL module in our framework. The model structure is described below and illustrated in Figure 5.7. Firstly, the model takes the observation from the environment as input. In our problem, the observation is the information of the gridworld and the PDDL state. The gridworld is directly encoded by a four-layer convolution neuron network with  $32 \times 64 \times 96 \times 128$  channel size, and the state that transfers into PDDL is encoded by a two-layer  $64 \times 64$  fully-connected network. Two types of encoded observation are concatenated and encoded by another two-layer  $64 \times 64$  fully-connected network. The output-encoded observation then uses as the input of the actor network and the critic network. The actor network executes the actions, and the critic network evaluates the actions.

### 5.3.2 Genetic Programming

Genetic programming is employed as a symbolic regressor for determining symbolic effect rules in the proposed methods, illustrated in Figure 5.8. We use the *gplearn*

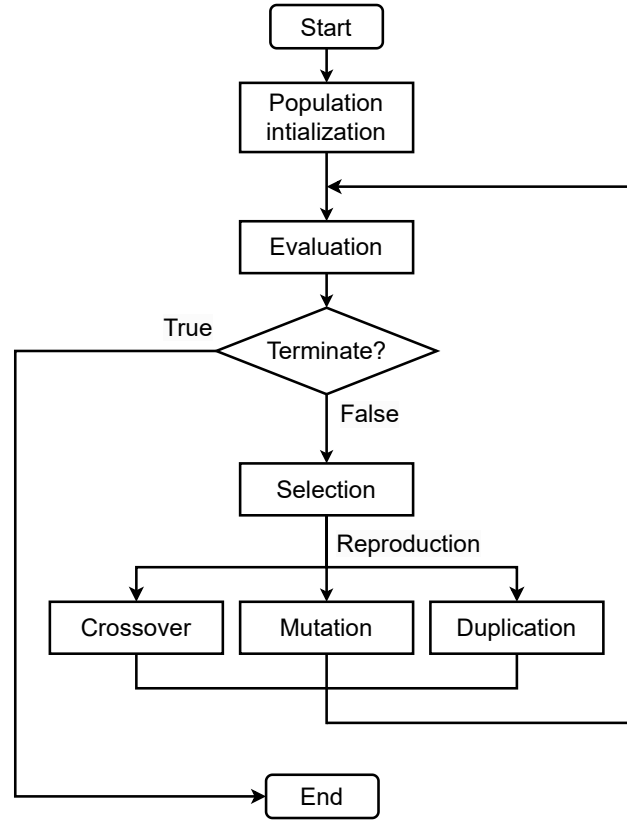


Figure 5.8: The procedure of genetic programming.

package for implementation and the parameter settings of are shown in Table 5.2. Given the action-effect linkage  $(a, \mathbf{E})$ , the transitions with action  $a$  are selected as the training data. For each effect variable  $e$  in  $\mathbf{E}$ , the algorithm's objective is to find the program that predicts the symbolic programs of  $e$  when executing the actions with the highest accuracy.

Each program is represented as an expression tree where input is the current state in the transition and output is the predicted value of  $e$ . The algorithm comprises several steps: initialization, evaluation, selection, crossover, and mutation. Initially, the population  $P$ , which is a set of programs, is randomly generated. Fitness evaluation is then performed on all programs; a subset of programs with the highest fitness values is selected. These programs serve as parents to produce offspring  $P_{\text{offspring}}$  through crossover and mutation mechanisms. Through iterative selection and production, the evolution of the population to discover the programs that best fit the given data. The evaluation metric used in genetic programming is

Table 5.2: The parameter setting of genetic programming. The parameter with two values indicates that the settings are different in two phases.

PARAMETERS	VALUE (FIRST/SECOND PHASE)
POPULATION SIZE	2000/2000
TOURNAMENT SIZE	200/200
GENERATIONS	20/10
P_CROSSOVER	0.6/0.6
P_SUBTREE_MUTATION	0.2/0.2
P_HOIST_MUTATION	0.1/0.1
P_POINT_MUTATION	0.05/0.05
MAX_SAMPLES	0.95/0.95
PARSIMONY_COEFFICIENT	0.0001/0.005
FUNCTION SET	$\{+, -, \times, \div\}/\{+, -, \times, \div\}$

the percentage of correct effect prediction which is shown below:

$$fitness(f_q) = \frac{\# \text{ of transitions with } (a, \mathbf{E}) \text{ consistent with } f_q}{\# \text{ of transitions with } (a, \mathbf{E})}, \quad (5.4)$$

where transition consistent with  $f_q$  means that the predicted effect  $f_q(s^t)$  from the current state  $s^t$  is consistent with the actual effect  $s_q^{t+1}$  given transition  $[s^t, a^t, s^{t+1}]$ .

According to the integer variable representation in this work, arithmetic operations  $\{+, -, \times, \div\}$  are selected as the function set. This implies that we assume that the effects are learnable using these operations. To be precise, the effects in the domain can be expressed by the function set and can be discovered through evolution. Also, to prevent bloat issues in which the program grows extremely larger to fit the data, the algorithm contains two phases: exploring and pruning. The best programs with the highest accuracy are determined in the exploring phase. Subsequently, in the pruning phase, the program with high parsimony while maintaining the same accuracy is selected as the output. The maximum number of generations in pruning phrases is the logarithm of the length of the best program in exploring phrases.

### 5.3.3 Reward Function

The reward function is decided by the number of steps for achieving the tasks and the maximum number of steps. The function are defined as follow:

$$R_{ext} = \frac{step_{max} - step}{step_{max}}, \quad (5.5)$$

where  $step_{max}$  is the maximum number of steps in the environment, and  $step$  is the steps the agent has done. Baseline agents only receive rewards when the tasks are achieved, while our framework provides intrinsic rewards for agents mentioned in Equation 4.5. Each time when the agent executes the correct critical actions, the reward are calculated where  $step$  is the current number of steps. The maximum steps in two environments are 25600.

## 5.4 Compared Algorithms

To assess the effectiveness of our framework, we have chosen several RL algorithms that share similarities with our approach. Firstly, we compare our work to the proximal policy optimization (PPO) [38] and Deep Q-Network (DQN) [29] algorithm as a standard RL baseline. Secondly, we evaluate our framework against other relevant works in the field. Our framework learns from demonstrations to acquire rules and employs intrinsic rewards to guide the agent. As a result, we select behavior cloning (BC) [35] and generative adversarial imitation learning (GAIL) [19] for comparison in terms of learning from demonstrations and rewarding impact-driven exploration (RIDE) [34] for comparison in terms of intrinsic rewards. In the following sections, we provide a detailed explanation and implementation of these algorithms.

### 5.4.1 Proximal Policy Optimization and Deep Q-Network

We select two classic DRL algorithms for evaluation: PPO and DQN. PPO was proposed by OpenAI in 2017 [38]. It is an on-policy reinforcement learning algorithm that aims to optimize policies for decision-making tasks. It utilizes the ideas from trust region policy optimization [37] which ensures that the policy update remains within a certain range to prevent drastic policy changes. In contrast to A2C, PPO incorporates a clip parameter that bounds the ratio between the new and old policies during the policy update. On the other hand, DQN developed by Google DeepMind [29] is a model-free reinforcement learning algorithm combining deep neural networks with Q-learning. In DQN, neural networks are used to approximate the Q function which evaluates the taken actions. The algorithm employs experience replay to store and randomly sample past experiences, which stabilizes learning and prevents overfitting. DQN has demonstrated impressive capabilities in handling high-dimensional state spaces, making it suitable for various complex tasks. Additionally, DQN utilizes a target network to reduce the risk of divergence during training.

For PPO, we use the implementation in *torch-ac* package. The architecture of the model is the same as our method illustrated in Figure 5.7. The batch size is 256; the entropy coefficient is 0.1; the value loss coefficient is 0.5; clipping epsilon is 0.2. For DQN, we use the implementation in *stable\_baselines3* package [33] for implementation and use the default settings of the packages.

### 5.4.2 Rewarding Impact-Driven Exploration

RIDE incorporates intrinsic rewards to incentivize the agent to take actions that have significant change. RIDE draws inspiration from the intrinsic curiosity module (ICM) which encourages the agent to explore the unseen states. Let the state embedding representation is  $\phi(s^t)$  at time  $t$ , ICM leverages the Euclidean distance between the predicted embedding and actual embedding of the next state  $L =$

$||\hat{\phi}(s^{t+1}) - \phi(s^{t+1})||_2^2$  to determine the intrinsic rewards. The forward model predicts  $\hat{\phi}(s^{t+1})$  and aims to minimize  $L$ , while the inverse model predict the action  $\hat{a}$  by  $\phi(s^t)$  and  $\phi(s^{t+1})$  and aims to minimize the cross entropy between  $\hat{a}$  and  $a$ . Instead of encouraging the agent to deviate from its predictions in ICM, RIDE uses the Euclidean distance between consecutive states  $L = ||\phi(s^{t+1}) - \phi(s^t)||_2$ , motivating the agent to actively explore and alter the current state within the environment. The intrinsic reward function of RIDE is defined as

$$R_{IDE}(s^t, a^t, s^{t+1}) = \frac{||\phi(s^{t+1}) - \phi(s^t)||_2}{\sqrt{N_{visited}(s^{t+1})}}, \quad (5.6)$$

where  $N_{visited}(s^t)$  is the number of visiting state  $s^t$  in this episode. We use the same configuration in the original paper [34]. The intrinsic reward coefficient is 0.1 and entropy cost is 0.0005, and the baseline cost is 0.5.

### 5.4.3 Behavior Cloning

BC is a supervised-learning approach used to imitate expert behavior. The model learns to map input states to corresponding actions based on the provided expert demonstrations. The objective of BC is to perform tasks with similar proficiency as the expert, leveraging the knowledge and skills encoded in the demonstrations.

However, the limitation of BC is the lack of exploration and adaptability. BC relies on expert demonstrations and lacks the ability to actively explore the environment or learn from its own experiences. This can hinder the agent's performance on unseen tasks. In contrast, our method also relies on the demonstrations without rewards to induce the model, assuming these demonstrations represent the desired behavior. The key distinction is that our framework is model-based, incorporating action schemata to enhance its capabilities. We use *stable\_baseline3* [33] for implementation and use the default settings of the packages.

#### 5.4.4 Generative Adversarial Imitation Learning

GAIL is an imitation learning approach developed in 2016 that combines the strengths of generative models and adversarial training. The objective of GAIL is to learn a policy that can imitate an expert’s behavior by jointly training a generator and a discriminator. Given states, the generator generates actions, while the discriminator learns to distinguish between the actions generated by the generator and those performed by the expert. This method incorporates the actor-critic method as the generator. The key advantage of GAIL is its ability to handle environments with sparse rewards. Since GAIL does not require explicit reward signals, it can learn from implicit feedback provided by the discriminator. This makes it particularly well-suited for tasks where expert demonstrations may be scarce or difficult to obtain. In the experiment, we use *stable\_baseline3* [33] for implementation. PPO is used as the generator, which settings are the same in Section 5.4.1. The demonstration batch size is 1024, and the replay buffer capacity is 2048.





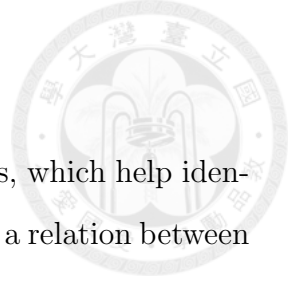
## Chapter 6

# Experiments and Discussions

In Chapter 4, we define the testing environments and provide implementation details. In this chapter, we present the empirical results conducted in the SWITCH and MINECRAFT environments. We begin with the preliminary experiment results of the framework are shown in Section 6.1, followed by a comparison of the framework’s performance with other algorithms in Section 6.2. We then demonstrate the framework’s capabilities for task generalization and skill generalization in Section 6.3. Finally, we discuss the limitations of our current work and offer insights for future improvements in Section 6.4.

### 6.1 Preliminary Experiment Results

In this section, we analyze the output of each process in the induction module and training module. In Section 6.1.1, we show that mutual information between actions and effects is a suitable metric for identifying linkages. In Section 6.1.2, we present the accuracy of inducing effect rules using genetic programming. In Section 6.1.3, we show the efficacy of employing pre-training and intrinsic rewards in the training module for navigating DRL agents.



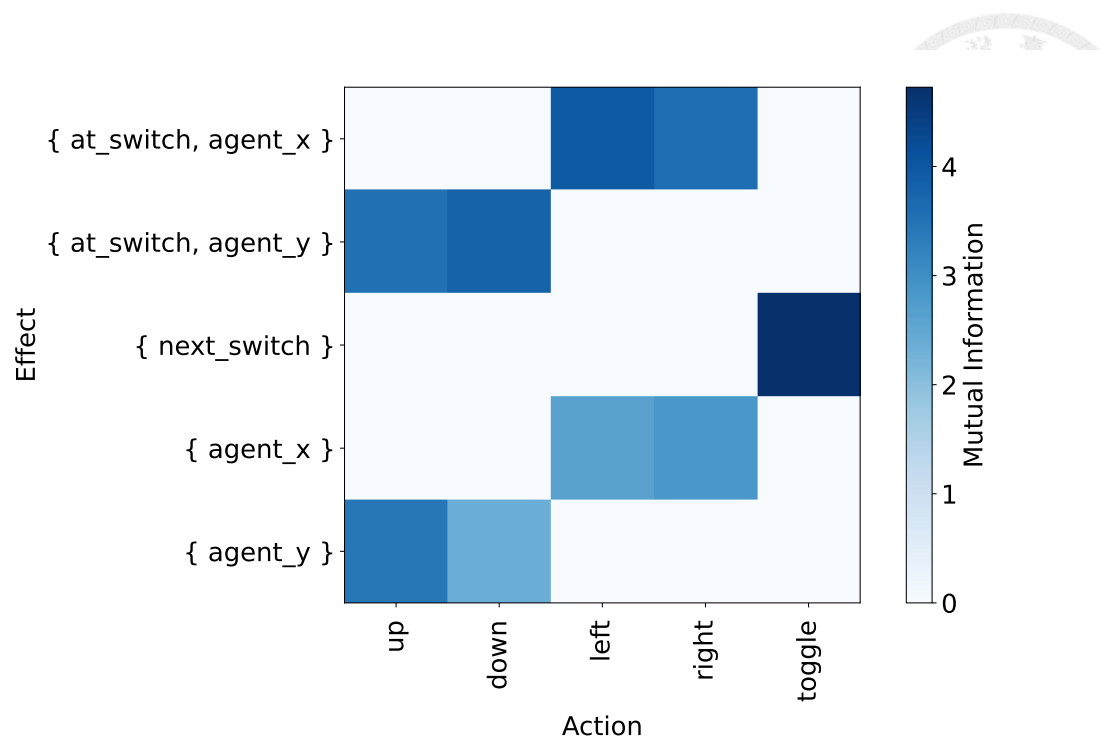
### 6.1.1 Action-Effect Linkage

In Section 4.2.2, we introduce the concept of action-effect linkages, which help identify the co-occurred effects and actions. We assume that if there is a relation between an action and its effects, they will co-occur with a higher probability. The degree of association between action-effect pairs can be identified by measuring the mutual information. Figure 6.1 presents the experimental results in MINECRAFT and SWITCH environments and shows the relationship between the logarithm of mutual information and action-effect linkages. The heat map visualizes the values of all action-effect pairs, with darker colors indicating higher values and stronger associations. For clarity, we only list the variables in effect variable space  $\mathbb{E}$  in MINECRAFT in Figure 6.1b, as these variables are used to induce the effect rules.

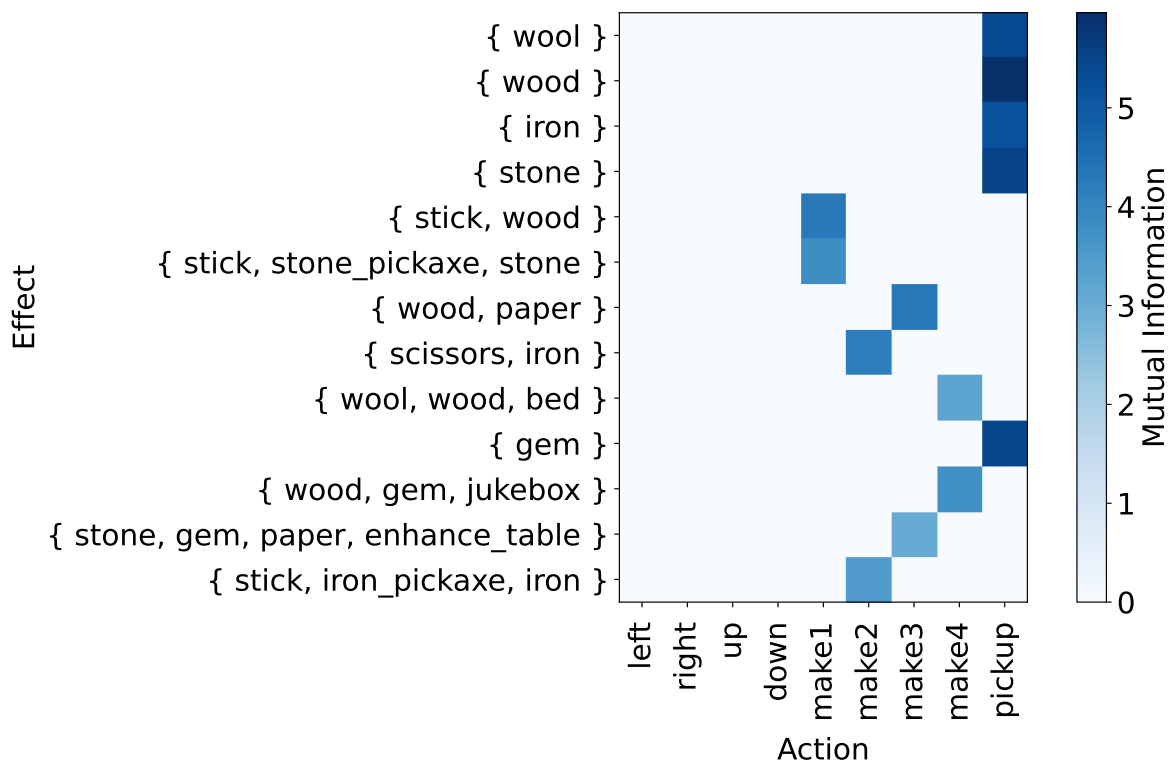
The results demonstrate a significant difference in mutual information between linked action and effect variables compared to unrelated ones. This means that when there is a relationship between action and effect variables, their co-occurrence increases, resulting in higher mutual information values. In Figure 6.1a, there is a linkage between `next_switch` and `toggle`, which indicates that the change of `next_switch` often caused by `toggle` in SWITCH environment. In Figure 6.1b, the higher values also distinguish the linkage and show which action affects the inventory in MINECRAFT environment.

### 6.1.2 Symbolic Regression

The proposed framework necessitates a robust symbolic regression module to generate the symbolic rules. In Section 5.3.2, we introduce genetic programming as symbolic regression for induction. Since genetic programming is a randomized search method, empirical results are shown to discuss the success rate of finding correct rules and how much demonstrations required to capture the symbolic rules.



(a) SWITCH



(b) MINECRAFT

Figure 6.1: Heatmap of mutual information.

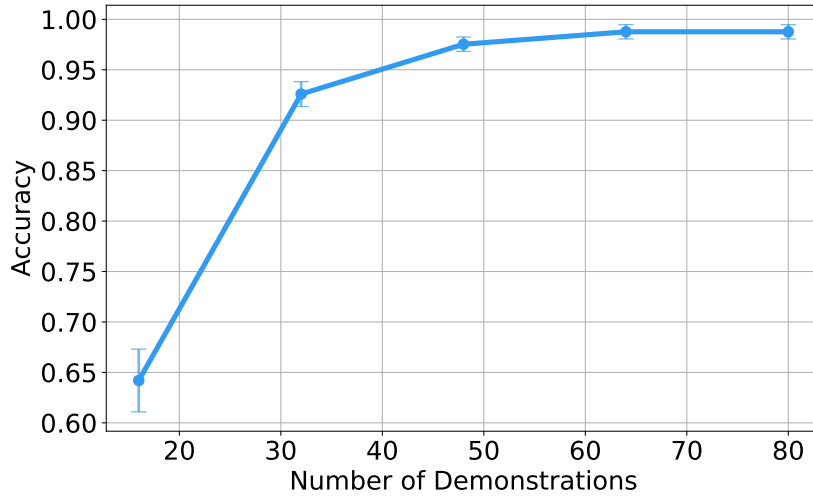


Figure 6.2: The accuracy of symbolic regression using genetic programming. The demonstrations are sampled from random subtasks, and the accuracy is the success rate of inducing 27 rules in MINECRAFT out of five runs.

The setting of the experiment is described as follows. In MINECRAFT environment, there are 27 effect rules listed in Table 5.1. We sample different numbers of demonstrations from random subtasks. The parameters of genetic programming are the same as the setting in Table 5.2. We test all the effect rules in each run and calculate the number of correct programs compared to the ground truth settings. The result is the average accuracy out of five runs shown in Figure 6.2. The result shows that the accuracy reaches 97.5% when sampling 64 demonstrations. Thus, we claim that the effect rules can be induced through genetic programming when a sufficient number of demonstrations are available.

### 6.1.3 Intrinsic Reward and Pre-Training

In this section, we provide experiment results to support the efficacy of intrinsic rewards and pre-training in enhancing the performance of DRL in SWITCH. Intrinsic rewards are given according to the critical action graph which encompasses the required subtasks of the task. When the agent correctly executes the critical action

that activates the switches, it receives a +1 reward. If there are  $n$  switches in the environment, the agent can only receive  $n$  times rewards. On the other hand, pre-training helps bootstrap the DRL agent to learn difficult tasks. If the task structures are known, the agent can effectively learn the tasks by pretraining in simpler environments. These techniques are incorporated into our framework, which serves as a guiding mechanism for the DRL agent to successfully complete tasks.

To evaluate the efficacy of the techniques, we design two experiments to evaluate the techniques. First, we compare the learning curves of DRL with and without intrinsic rewards in the 4-SWITCHES task. The results illustrated in Figure 6.3 demonstrate that incorporating intrinsic rewards significantly improves training efficiency. The agent trained with intrinsic rewards outperforms the baseline method receiving 95% of average extrinsic rewards. Second, we examine the agent's performance with pre-training and intrinsic rewards in 16-SWITCHES. The results are shown in Figure 6.4. In this case, although the agent is pre-trained in 4-SWITCHES, it still fails to learn the task without intrinsic rewards and does not receive any rewards during training, while the agent with intrinsic rewards receives higher average rewards. This result highlights the significance of intrinsic rewards in facilitating efficient task learning.

Then we compare the training efficiency between two different approaches with intrinsic rewards: The first approach pre-trains the agent in 4-SWITCHES, and the second approach only provides intrinsic rewards in 16-SWITCHES. Noting that 4-SWITCHES serves as a preliminary task that the agent must successfully accomplish the goal of 4-SWITCHES as the subgoal of 16-SWITCHES. Although the approach with both pre-training and providing intrinsic rewards is more effective than only intrinsic rewards, the agent fails to learn the tasks with only pre-training. Also, the task difficulties depend on domains based on the complexity and the configuration of environments. Therefore, it often requires hand-craft tasks with different difficulties for training, which is against our assumption of automatic induction and training. Thus, we select intrinsic rewards as the role of guiding DRL agents.

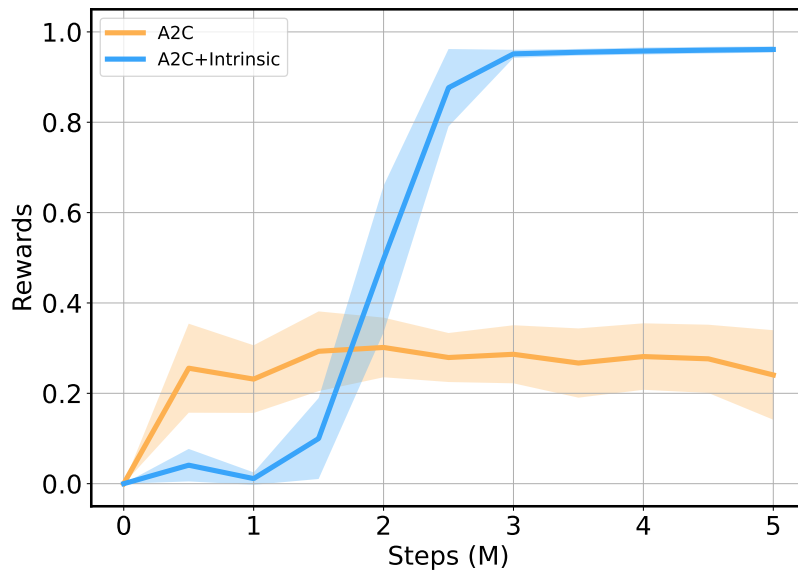


Figure 6.3: The training performance with intrinsic rewards in 4-SWITCHES environment. The baseline is A2C without intrinsic rewards.

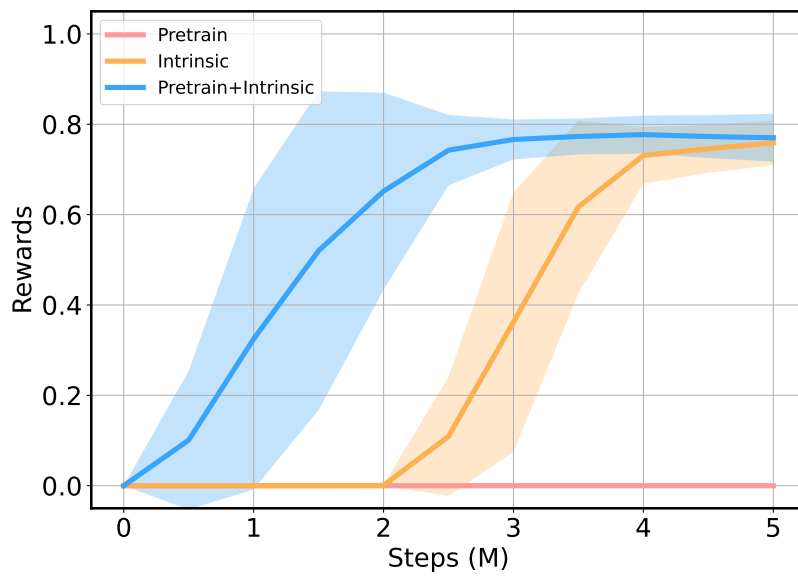


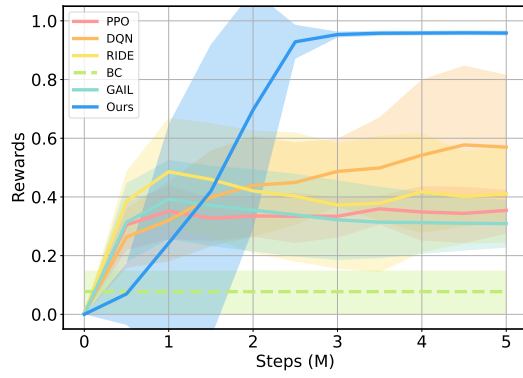
Figure 6.4: The training performance comparing pretraining and intrinsic rewards in 16-SWITCHES environment.

## 6.2 Experiment Results on Performance

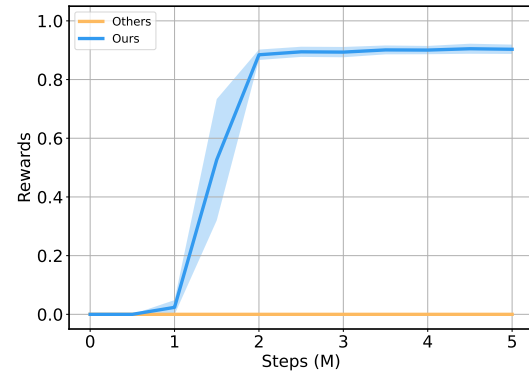
This section presents the experiment results on SWITCH and MINECRAFT. Results in SWITCH demonstrate the ability to learn hierarchical tasks, while MINECRAFT shows the performance of learning compositional tasks and multiple tasks. We run the experiments over five random seeds and show the mean of the training curves over 5M steps out of five runs. In each plot, the solid lines are the mean of training curves, and the shaded region is the standard deviation.

### 6.2.1 Switch

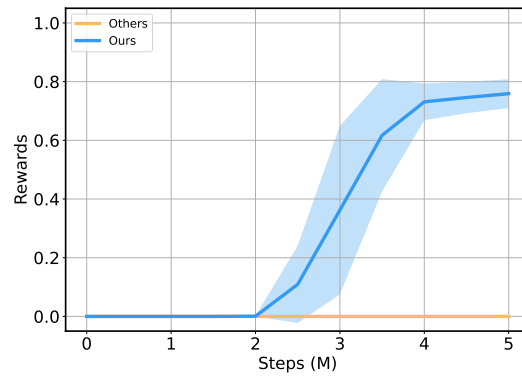
We evaluate four tasks in SWITCH, The tasks 4-SWITCHES, 8-SWITCHES and 16-SWITCHES evaluate the performance of the algorithm when the hierarchy of tasks increases. Besides, 4-SWITCHES-4-DISTRACTORS and 4-SWITCHES-4-ROOMS are set to test the performance in various environments. For each run, we collect 20 demonstrations for BC, GAIL, and our method. The results are shown in Table 6.1 and Figures 6.5 and 6.6. Our method improves the average rewards and training efficiency compared to the other algorithms, especially on 16-SWITCHES tasks. In 4-SWITCHES, the proposed method achieves  $96 \pm 01\%$  of rewards, while others achieve less than 34% of rewards. The difference in performance is more significant in complicated tasks. While other methods failed to learn the tasks, our method achieves  $90 \pm 01\%$  and  $75 \pm 03\%$  of rewards in 8-SWITCHES and 16-SWITCHES, respectively. In addition, in 4-SWITCHES-4-DISTRACTORS and 4-SWITCHES-4-ROOMS, all methods have lower performance compared with 4-SWITCHES due to the distractors and the walls, while our method still reaches  $91 \pm 05\%$  average rewards, which outperforms other methods. In summary, the result shows that our method improves training efficiency, especially when the tasks are more difficult to reach the goal.



(a) 4-SWITCHES

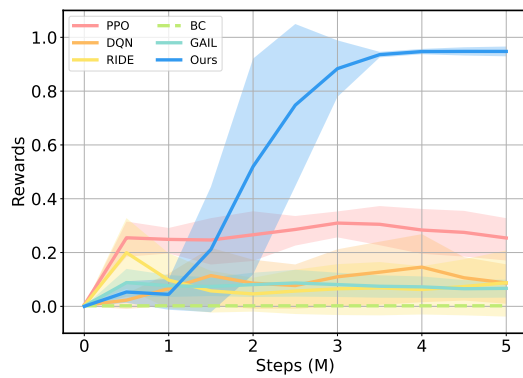


(b) 8-SWITCHES

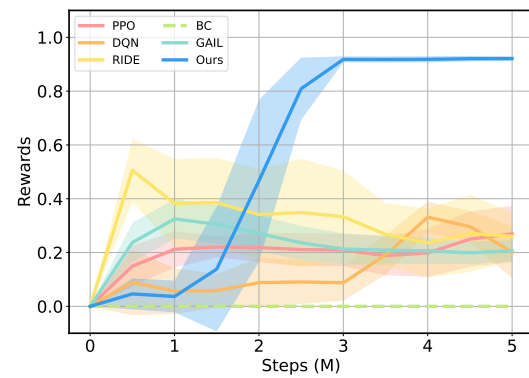


(c) 16-SWITCHES

Figure 6.5: Training performance in SWITCH with different numbers of switches.



(a) 4-SWITCHES-4-DISTRACTORS



(b) 4-SWITCHES-4-ROOMS

Figure 6.6: Training performance in SWITCH with distractors and rooms.



Table 6.1: Performance in SWITCH environment after 5M steps.

Task	PPO	DQN	RIDE	BC	GAIL	Ours
4-SWITCHES	.34±.07	.56±.05	.32±.18	.07±.07	.30±.08	<b>.96±.01</b>
8-SWITCHES	.00±.00	.00±.00	.00±.00	.00±.00	.00±.00	<b>.90±.01</b>
16-SWITCHES	.00±.00	.00±.00	.00±.00	.00±.00	.00±.00	<b>.75±.03</b>
4-SWITCHES-4-DISTRACTORS	.24±.08	.08±.05	.08±.12	.00±.03	.06±.04	<b>.95±.02</b>
4-SWITCHES-4-ROOMS	.27±.10	.20±.10	.26±.09	.00±.00	.21±.04	<b>.92±.01</b>

Table 6.2: Performance in MINECRAFT environment after 5M steps.

Task	PPO	DQN	RIDE	BC	GAIL	Ours
IRON	<b>.84±.01</b>	.30±.10	.75±.09	.00±.03	.17±.12	<b>.82±.02</b>
ENHANCETABLE	.63±.09	.16±.04	.20±.04	.00±.00	.03±.02	<b>.73±.05</b>
MULTIPLE	.46±.15	.15±.07	.50±.06	.00±.06	.45±.14	<b>.74±.03</b>

## 6.2.2 Minecraft

This section shows the empirical result about MINECRAFT Environment. In MULTIPLE, we randomly select a task as the goal at the beginning of each episode. In this scenario, the task of each episode is selected uniformly at random. In addition, we select the task of getting an iron and the task of getting a enhance table as the metrics named IRON and ENHANCETABLE, respectively. In each run, 64 demonstrations with random goals are collected for our methods and imitation learning approaches. This configuration demonstrates that our method is able to induce critical actions from shared subtasks and construct the graph based on the desired goal. The results are shown in Figure 6.7 and Table 6.2. In simple tasks such as IRON, PPO and RIDE reach high performance as well as our method. However, in difficult tasks such as EHANCETABLE, only our method outperforms other methods with 73±05% of average rewards. In addition, in MULTIPLE, our method receives 74±03% rewards than other methods.

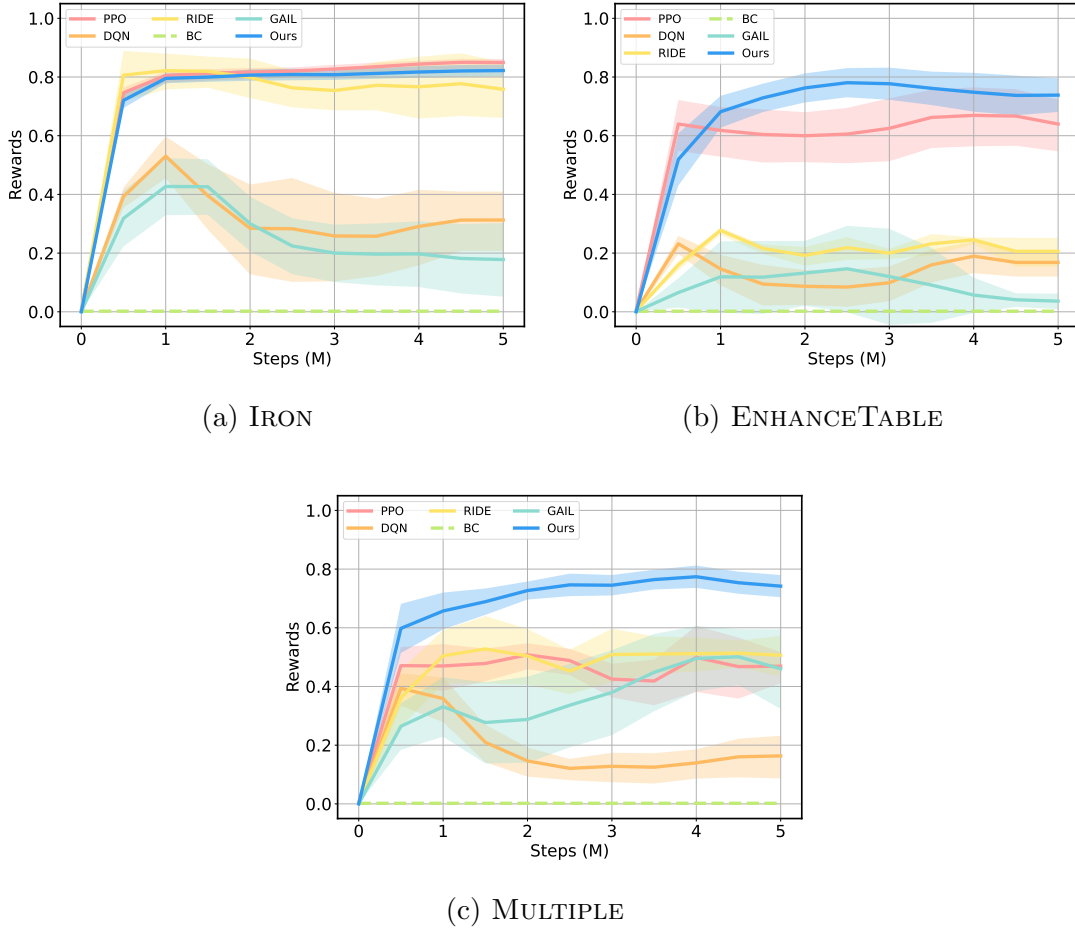


Figure 6.7: Training performance in MINECRAFT environment.

### 6.3 Generalizability

In this section, we demonstrate the generalizability of our method. Our method leverages demonstrations to construct the critical action model. This model defines subtasks through critical actions consisting of preconditions, effects, and MDP actions. These components are compositional, providing a level of generalizability to the agents. For comparison, we select BC and GAIL, both of which also learn from demonstrations. In Sections 6.3.2 and 6.3.3, we will show the result of performing task generalization and skill generalization.

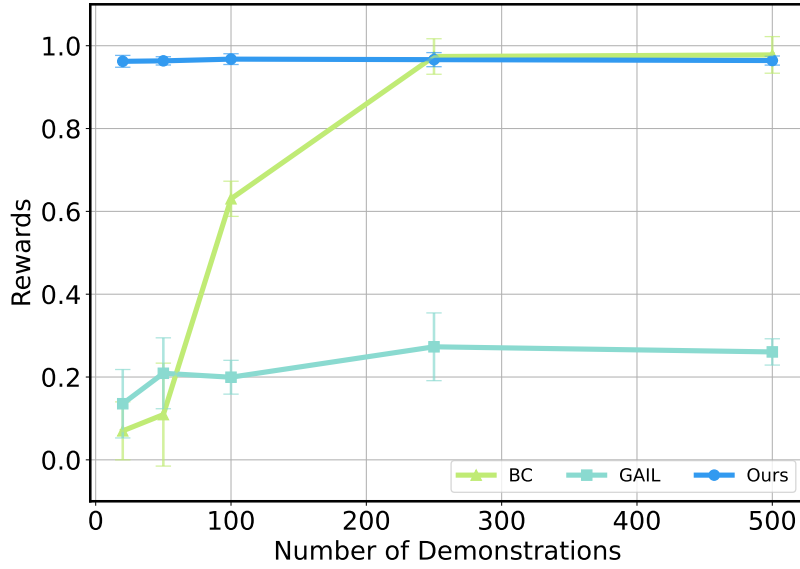


Figure 6.8: The performance of BC, GAIL and our methods with different demonstrations after 5M steps in 4-SWITCHES-INCREMENTAL.

### 6.3.1 Preliminaries

To assess generalizability, we assume that a sufficient number of demonstrations are available to train the agent within the original domain. In this section, we examine the performance of BC, GAIL, and our method using varying numbers of demonstrations. The results shown in Figure 6.8 demonstrate that BC achieves an average reward of 97% with 250 demonstrations. Our approach, on the other hand, achieves a comparable average reward of 96% with only 20 demonstrations for induction. In contrast, GAIL requires more than 500 demonstrations to achieve the same performance level. For equitable comparison, we employ 250 demonstrations for all methods in the subsequent experiments. Our focus lies in contrasting the performance differences between the original and shifted domains. In Sections 6.3.2 and 6.3.3, we will demonstrate the generalizability of our method at task and skill levels, while other learning-from-demonstration methods failed to generalize the knowledge learned from the original domain.

### 6.3.2 Task Generalization

Task generalization enables the agents to learn the new task with the prior knowledge of learned tasks. Our method employs the critical action model, which is generalizable at the task level by constructing new structures from known critical actions. Thus, the proposed framework can provide task generalization if the rules of the environment remain unchanged. On the other hand, BC utilizes supervised learning to imitate the given demonstration, which limits the agent’s ability to generalize knowledge when encountering unseen tasks, while GAIL utilizes generative adversarial networks and training agents by distinguishing between actions generated by the policy network and actions from expert demonstrations. These model-free imitation learning approaches suffer from the issue of distribution shift. If the training domain varies from the demonstration domain, the agent fails to learn the tasks.

In our study, we define a shifted domain as a scenario where the tasks share the same critical actions but have different task structures that have not been encountered before. This means that although the actions required to accomplish the tasks are identical, the way they are organized or sequenced in each task is unfamiliar to the agent. In our framework, the training module can reason the required critical actions and compose the task structure of unseen tasks based on few-shot demonstrations.

To demonstrate these properties, we evaluate in 4-SWITCHES-INCREMENTAL and its shifted domains. We collect 250 demonstrations in 4-SWITCHES-INCREMENTAL for all methods, train 300 epochs for BC, and run 5M steps for GAIL and our methods. The results in Table 6.3 demonstrate that the performance of BC and GAIL drops in the shifted domain, whereas our model outperforms other methods in the shifted domain. Our method induces critical action `turn_on`, and the framework constructs different critical action graphs with the critical actions, which enables the agent adapts to the shifted domain.

Table 6.3: The generalization performance in four tasks.

Task	BC	GAIL	Ours
4-SWITCHES-INCREMENTAL (demonstrations)	<b>.95±.07</b>	.30±.08	<b>.96±.01</b>
8-SWITCHES-INCREMENTAL	.26±.07	.10±.02	<b>.90±.02</b>
4-SWITCHES-4-DISTRACTORS-INCREMENTAL	.00±.00	.10±.07	<b>.95±.02</b>
4-SWITCHES-4-DISTRACTORS-ODD	.00±.00	.11±.06	<b>.95±.01</b>

### 6.3.3 Skill Generalization

In Section 4.2.3, we introduce genetic programming for reasoning symbolic rules. One advantage of employing evolutionary computation is its adaptability. We define a variant critical action  $\phi$  from  $\psi$  where  $\mathbf{P}_\phi = \mathbf{P}_\psi$  and  $\mathbf{E}_\phi = \mathbf{E}_\psi$  while  $F_\phi^{\mathbf{P}}$  or  $F_\phi^{\mathbf{E}}$  changes. If a critical action  $\psi$  varied, the induced symbolic programs and the population are able to evolve and adapt to new rules. Since  $\mathbf{P}_\phi$  and  $\mathbf{E}_\phi$  are known, the procedure starts from inducing effect rules  $F_\phi^{\mathbf{E}}$ .

We focus on few-shot demonstration generalization. That is, given sufficient demonstrations of original tasks and few-shot demonstrations of varied tasks, whether the agent can adapt to new skills and learn the tasks. We collect 250 demonstrations in 4-SWITCHES-INCREMENTAL and four-shot demonstrations in 4-SWITCHES-ODD for BC and GAIL. We induce the new critical action starting from the step of determining effect rules in Section 4.2.3. For GAIL and our method, agents are trained in 4-SWITCHES-4-DISTRACTORS-ODD configuration. The result in Table 6.3 shows that the performance of BC and GAIL decrease in 4-SWITCHES-4-DISTRACTORS-ODD. In contrast, our method induces new critical actions from few-shot demonstrations and provides intrinsic rewards when the agent activates the switch in an odd order.

## 6.4 Discussions

This section discusses the framework and its performance in different aspects. In Section 6.4.1, another approach to constructing task structures using induction from demonstrations is discussed. In Section 6.4.2, we contrast with other approaches focusing on learning from demonstrations and hierarchical task learning. These methods are not compared in the experiments due to the different scenarios. In Section 6.4.3, the limitations of the framework are also discussed.

### 6.4.1 Extracting Task Structures from Demonstrations

In some cases, when the environment is partially observable, the critical action model may be incomplete, Especially when the preceding critical action's effects and succeeding one's preconditions are not directly associated, leading to the failure of deducing the critical action model due to a lack of information. One solution to this issue is extracting the critical action sequences from demonstrations and constructing critical action graphs from the sequences illustrated in Figure 6.9. Given the critical action model, we can label the critical actions and discover the sequential order of the critical actions. Some related works develop approaches to induce the task structure from action sequences, including automata [47, 13] and hierarchical task network [17], which can also be utilized in our frameworks. However, these methods rely on the demonstration quality and can not generalize to unseen tasks.

### 6.4.2 Comparison with Other Methods

In the previous section, we compare the performance between our methods and other studies. However, these methods have a limitation that may fail in different scenarios. In this section, we furtherly discuss the assumption and the limitation of these methods, including exploration-based intrinsic rewards and imitation learning.

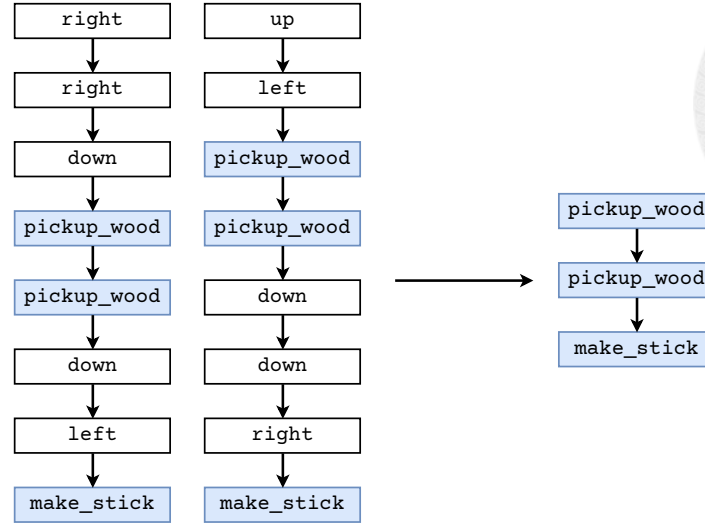


Figure 6.9: The illustration of inducing task structures from critical action sequences.

## Exploration-Based Intrinsic Rewards

RIDE encourages the agent to explore the environment by rewarding the behavior that significantly changes the environment. Its predecessor ICM incentivizes the behavior of exploring unexpected states. These methods expect that exploration improves the effectiveness of learning. However, sometimes exploration may hinder the agent from achieving the goal. For instance, exploration may lead to frequently deactivating the switches in SWITCH. In the environment with a long episode length, The agent will urge to pursue intrinsic reward instead of reaching the goal [48]. These methods also suffer from exploration-exploitation dilemmas that the hyperparameter of exploration-exploitation balance needs to be hand-crafted [24].

In contrast, our method not only incentivizes the unexpected or significant effects but the correct effects. The framework has the knowledge of critical action graphs and provides the reward when the agent executes specific critical actions. The number of execution are also included in critical action graphs. This enable the framework stop providing reward if the agent tends to gain short-term intrinsic rewards without achieving the long-term goal.

## Imitation Learning

In this study, we select BC and GAIL as the benchmarks of imitation learning. These techniques train an agent to directly imitate desired behaviors from state-action pair in demonstrations. The process of BC involves collecting expert demonstrations and training agents to learn the mapping between states and actions of demonstrations. The model is trained to approximate the expert's behavior by minimizing the discrepancy between generated and expert actions without explicitly modeling the underlying decision-making process. GAIL leverages the concept of Generative Adversarial Networks, where the generator generates actions imitating the expert's actions, while the discriminator aims to differentiate between the two.

However, BC and GAIL suffer from the issue of distribution shift caused by the sampling bias from demonstrations. Also, they lack the ability to adapt to other unseen tasks in the same domain due to the limited data that some situations are absent in demonstrations. Sufficient demonstrations in testing domains are necessary for these imitation learning methods. Therefore, despite few-shot demonstrations in the shifted domains given, the agent using these techniques failed to achieve the tasks in these domains. In contrast to BC and GAIL, our proposed method addresses the limitations by employing model-based learning, our framework induces the model in terms of critical actions, which enables the DRL agents to possess generalizability by re-inducing rules and task structures. In Sections 6.3.2 and 6.3.3, we assess the generalizability in both task and skill levels. Under the specific scenario, our framework is capable to adapt to unseen tasks.

### 6.4.3 Limitation

While the improvement of performance is demonstrated in this chapter, there are some unaddressed issues in this thesis. First, the method is tested in a deterministic environment, and the configuration of the logical formula in critical actions does not consider the probability of multiple outcomes. This can be solved by introducing



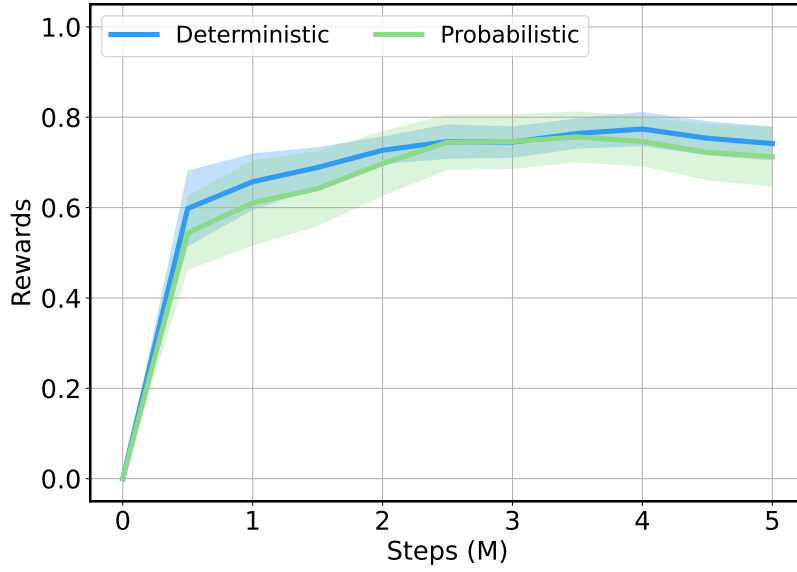


Figure 6.10: The training performance of our framework in MINECRAFT-MULTIPLE with the deterministic and probabilistic environments.

a probabilistic model for abstraction. Some researchers have delved to develop probabilistic rules in planning domains [31, 3]. In this work, we focus on introducing the idea of critical actions. Hence we opt to use deterministic models. Nevertheless, the proposed method works in the probabilistic environment at the execution level. In this work, we use the DRL module for execution, which has the capability of handling stochastic environments. To show this property, we train the agent in the MINECRAFT-MULTITASK with a probabilistic moving configuration. When the agent executes `up`, `down`, `left`, `right`, it moves to desired direction with a probability of 0.8, but turns to the left or right of the desired direction with each probability of 0.1. The result in Figure 6.10 shows that the DRL agent can still learn the task in the probabilistic domain.

Second, although the empirical results show that each module in the framework generates optimal output with high probability, incorrect inference in some processes may lead to error propagation. For instance, if incorrect effects are found during symbolic regression, the critical action may mislead the task structure construction and therefore fail to provide correct rewards. In induction modules, action-effect

linkage extraction and precondition determination employed deterministic methods, and these methods were affected by the amount and distribution of demonstrations. On the other hand, evolutionary computation applied in effect rules determination is a stochastic method that with some probability the outcome is not optimal. This situation can be avoided by increasing the population in the algorithm or sampling several times to acquire more accurate results.

Last but not least, the induction of critical actions is offline and relies on demonstrations. Although we have demonstrated adaptability using offline few-shot demonstrations, the framework does not leverage the online data for adaptation. Consider the zero-shot generalization scenario, the agent should acquire the ability to infer new rules and optimize the critical action graph during the interaction with the environment. Therefore, there is a potential for further improvement by incorporating online data for real-time adaptation.

In summary, we evaluate the performance of our framework in various domains and compared it with other approaches that use demonstrations and intrinsic rewards for learning. We discussed the strengths and limitations of different algorithms, highlighting the drawbacks of existing methods and emphasizing the advantages of our proposed framework in these scenarios. Despite the successes of our method, we also acknowledge its limitations and future work should focus on addressing these challenges.

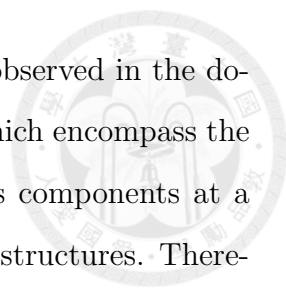


## Chapter 7

# Conclusion and Future Work

In this study, we present a framework to address goal-directed MDP problems by integrating DRL and planning techniques. Our framework involves the representation of symbolic knowledge as critical action graphs and the procedure of automatically extracting knowledge from learnable tasks. Specifically, we elaborate on the process of extracting knowledge from demonstrations within the DRL domain, constructing a critical action model within the planning domain, and subsequently leveraging this knowledge back into the DRL domain by providing intrinsic rewards. The combination of inductive learning and deductive learning enables the agent to perform high-level abstraction and low-level execution. In addition, the proposed framework shows generalizability at task and skill levels due to the compositionality of the critical action model. At the task level, the training module deduces unseen tasks and constructs critical action graphs from known critical actions which leads to task generalization; At the skill level, the induction module infers new effect rules from few-shot demonstration while the linkages of action-effect pairs are known.

Evolutionary computation for symbolic regression provides adaptability in dynamically changing environments. Under the mechanism of evolutionary computation, the agent is capable of adapting to new environments by evolving and generating new symbolic knowledge from a pool of candidate programs. The symbolic



programs are considered atoms that capture fundamental rules observed in the domains. These rules are then combined to form critical actions, which encompass the effects and preconditions. The critical actions which operate as components at a higher level of abstraction constitute the building blocks of task structures. Therefore, diversity of rules is essential for fostering flexible intelligence, as it enables versatility in problem-solving, and evolutionary computation inspired by natural evolution aligns with these principles by promoting the generation and preservation of diverse and varied concepts.


Some approaches can be improved in future works. We demonstrated that intrinsic rewards are one of the intuitive methods to facilitate the agent's training efficiency. The design emphasizes the cooperation of two independent modules, one for abstraction and planning and the other for execution and adaptation. However, using traditional neural networks to learn multi-tasks is not the most efficient approach. To maximize the utility of the symbolic knowledge, hierarchical reinforcement learning which alternate options when executing different subtasks is more suitable in this case. The critical action provides a guide to pre-train the options by setting initial states in which the precondition is satisfied and the goal is to make the effect occur. On the other hand, the curriculum learning scenario can also be adopted with this framework. Given the critical action graphs, tasks can be generated by selecting part of the graphs and setting the top preconditions satisfied as the initial state and the bottom effect as the desired goal.

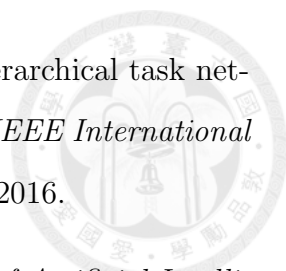
By representing knowledge as critical actions and employing critical action graphs, we provide a structured and organized means of capturing and utilizing symbolic knowledge within DRL. The procedures of subtask decomposition combine planning and DRL, leading to effective learning in goal-directed tasks. The compositionality of the critical action model enables different levels of generalization, indicating its potential to address a wide range of general problems. Our work offers a holistic perspective to effectively handle general goal-directed decision-making problems with the integration of inductive and deductive learning.




# Bibliography

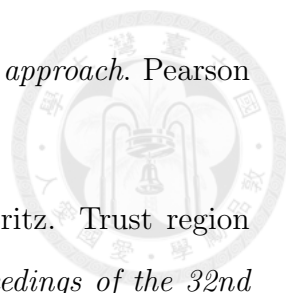
- [1] D. Abel, D. Arumugam, L. Lehnert, and M. Littman. State abstractions for lifelong reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning*, pages 10–19. PMLR, July 2018. ISSN: 2640-3498.
- [2] J. Andreas, D. Klein, and S. Levine. Modular multitask reinforcement learning with policy sketches. In *International Conference on Machine Learning*, pages 166–175, 2017.
- [3] A. Arora, H. Fiorino, D. Pellier, M. Métivier, and S. Pesty. A review of learning planning action models. *The Knowledge Engineering Review*, 33:e20, 2018.
- [4] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.
- [5] R. S. Aylett and G. J. Petley. AI planning: Solutions for real world problems. *Knowledge-Based Systems*, 13(2):61–69, 2000.
- [6] T. Back, U. Hammel, and H.-P. Schwefel. Evolutionary computation: Comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1(1):3–17, 1997.
- [7] E. A. Brooks, J. Rajendran, R. L. Lewis, and S. Singh. Reinforcement learning of implicit and explicit control flow in instructions. In *International Conference on Machine Learning*, pages 1082–1091, 2021.

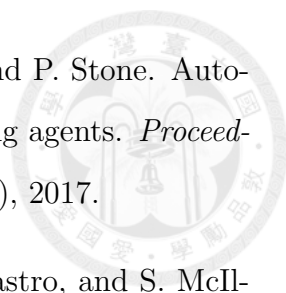
- 
- [8] R. W. Byrne and A. E. Russon. Learning by imitation: A hierarchical approach. *Behavioral and Brain Sciences*, 21(5):667–684, 1998.
- [9] L. C. Cobo, P. Zang, C. L. I. Jr, and A. L. Thomaz. Automatic state abstraction from demonstration. In *International Joint Conference on Artificial Intelligence*, volume 22, page 1243, 2011.
- [10] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov. OpenAI baselines. <https://github.com/openai/baselines>, 2017.
- [11] C. Florensa, Y. Duan, and P. Abbeel. Stochastic neural networks for hierarchical reinforcement learning. 2017. arXiv:1704.03012 [cs].
- [12] D. Furelos-Blanco, M. Law, A. Jonsson, K. Broda, and A. Russo. Induction and exploitation of subgoal automata for reinforcement learning. *Journal of Artificial Intelligence Research*, 70:1031–1116, 2021.
- [13] D. Furelos-Blanco, M. Law, A. Russo, K. Broda, and A. Jonsson. Induction of subgoal automata for reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3890–3897, 2020.
- [14] M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. Smith, Y. Sun, and D. Weld. PDDL - the planning domain definition language. *Technical Report*, 1998.
- [15] J. J. Grefenstette, C. L. Ramsey, and A. C. Schultz. Learning sequential decision rules using simulation models and competition. *Machine Learning*, 5(4):355–381, Oct. 1990.
- [16] L. Guan, S. Sreedharan, and S. Kambhampati. Leveraging approximate symbolic models for reinforcement learning via skill diversity, June 2022. arXiv:2202.02886 [cs].

- 
- [17] B. Hayes and B. Scassellati. Autonomously constructing hierarchical task networks for planning and human-robot collaboration. In *2016 IEEE International Conference on Robotics and Automation*, pages 5469–5476, 2016.
- [18] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.
- [19] J. Ho and S. Ermon. Generative adversarial imitation learning. *Advances in neural information processing systems*, 29, 2016.
- [20] J. Hoffmann. FF: the fast-forward planning system. *AI magazine*, 22(3):57–57, 2001.
- [21] R. T. Icarte, T. Klassen, R. Valenzano, and S. McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning*, pages 2107–2116, 2018.
- [22] R. T. Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith. Reward machines: exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research*, 73, 2022.
- [23] T. D. Kulkarni, K. R. Narasimhan, A. Saeedi, and J. B. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.
- [24] P. Ladosz, L. Weng, M. Kim, and H. Oh. Exploration in deep reinforcement learning: A survey. *Information Fusion*, 85:1–22, 2022.
- [25] lcsவில்லெம்ஸ். Pytorch actor-critic deep reinforcement learning algorithms: A2C and PPO. <https://github.com/lcsவில்லெம்ஸ்/torch-ac>, 2022.
- [26] A. Liu, S. Sohn, M. Qazwini, and H. Lee. Learning parameterized task structure for generalization to unseen entities. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 7534–7541, 2022.

- 
- [27] W.-Y. Loh. Classification and regression trees. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 1(1):14–23, 2011.
- [28] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016. arXiv:1602.01783 [cs].
- [29] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.
- [30] T. M. Moerland, J. Broekens, A. Plaat, C. M. Jonker, et al. Model-based reinforcement learning: A survey. *Foundations and Trends® in Machine Learning*, 16(1):1–118, 2023.
- [31] H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.
- [32] S. Pateria, B. Subagdja, A.-h. Tan, and C. Quek. Hierarchical reinforcement learning: A comprehensive survey. *ACM Computing Surveys*, 54(5):1–35, 2022.
- [33] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.
- [34] R. Raileanu and T. Rocktäschel. RIDE: Rewarding impact-driven exploration for procedurally-generated environments. In *International Conference on Learning Representations*, 2020.
- [35] S. Ross, G. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.



- 
- [36] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Pearson Education, Inc., 3rd edition, 2010.
- [37] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In F. Bach and D. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, 2015. PMLR.
- [38] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017. arXiv:1707.06347 [cs].
- [39] C. E. Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.
- [40] T. Silver and R. Chitnis. PDDLgym: gym environments from PDDL problems. In *International Conference on Automated Planning and Scheduling (ICAPS) PRL Workshop*, 2020.
- [41] S. Sohn, J. Oh, and H. Lee. Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies. *Advances in neural information processing systems*, 31, 2018.
- [42] S. Sohn, H. Woo, J. Choi, and H. Lee. Meta reinforcement learning with autonomous inference of subtask dependencies, 2020. arXiv:2001.00248 [cs, stat].
- [43] S. Sohn, H. Woo, J. Choi, l. qiang, I. Gur, A. Faust, and H. Lee. Fast inference and transfer of compositional task structures for few-shot task generalization. In *Uncertainty in Artificial Intelligence*, pages 1857–1865, 2022.
- [44] S. Sukhbaatar, E. Denton, A. Szlam, and R. Fergus. Learning goal embeddings via self-play for hierarchical reinforcement learning, 2018. arXiv:1811.09083 [cs, stat].
- [45] S.-H. Sun, T.-L. Wu, and J. J. Lim. Program guided agent. In *International Conference on Learning Representations*, 2020.

- 
- [46] M. Svetlik, M. Leonetti, J. Sinapov, R. Shah, N. Walker, and P. Stone. Automatic curriculum graph generation for reinforcement learning agents. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), 2017.
- [47] R. Toro Icarte, E. Waldie, T. Klassen, R. Valenzano, M. Castro, and S. McIlraith. Learning reward machines for partially observable reinforcement learning. *Advances in Neural Information Processing Systems*, 32, 2019.
- [48] S. Wan, Y. Tang, Y. Tian, and T. Kaneko. Deir: Efficient and robust exploration through discriminative-model-based episodic intrinsic rewards. In *International Joint Conference on Artificial Intelligence*, 2023.
- [49] Z. Wang, S. Cai, A. Liu, X. Ma, and Y. Liang. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents, 2023. arXiv:2302.01560 [cs].
- [50] Z. Xu, B. Wu, A. Ojha, D. Neider, and U. Topcu. Active finite reward automaton inference and reinforcement learning using queries and counterexamples. In A. Holzinger, P. Kieseberg, A. M. Tjoa, and E. Weippl, editors, *Machine Learning and Knowledge Extraction*, pages 115–135. Springer International Publishing, 2021.
- [51] C. A. Y. Blandin, L. Proteau. On the cognitive processes underlying contextual interference and observational learning. *Journal of Motor Behavior*, 26(1):18–26, 1994.



## Appendix A

### Task Structure of Test Problems

The appendix shows the critical action graphs of tasks IRON and ENHANCETABLE in MINECRAFT. Due to the complexity of graph, we only illustrate partial graphs which include all nodes with a depth of 2 or less from the goal.

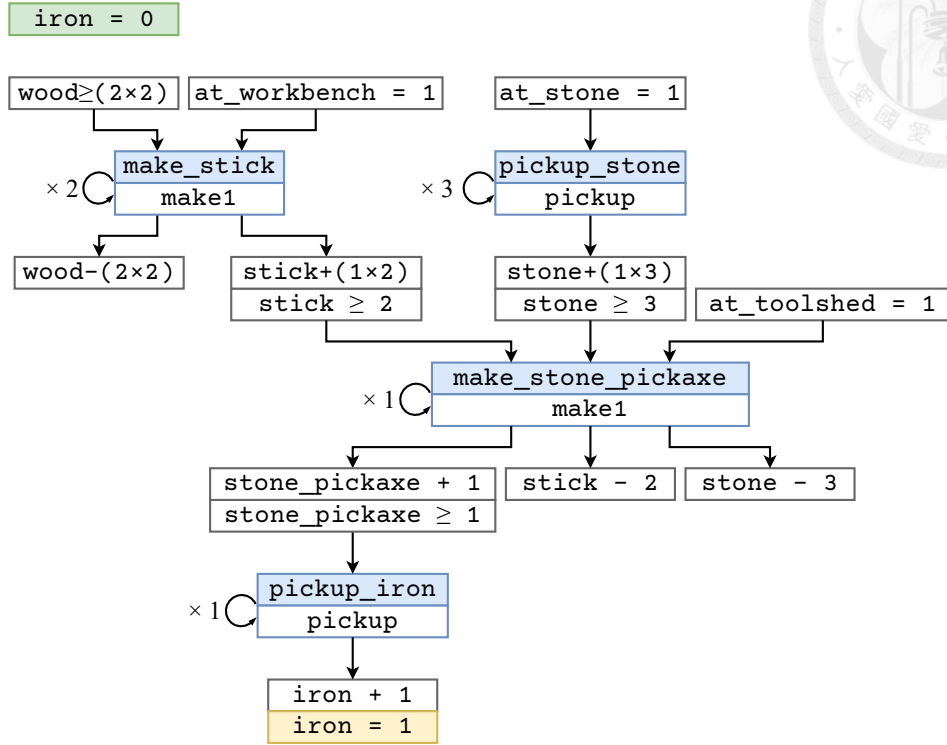


Figure A.1: Critical action graph of IRON

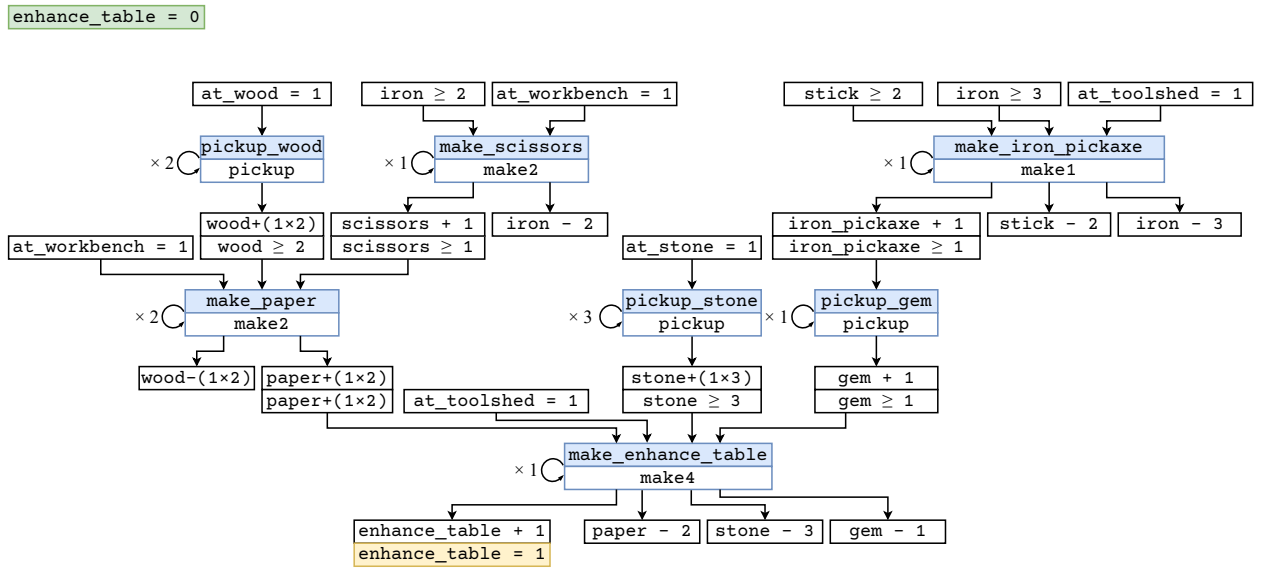


Figure A.2: Critical action graph of ENHANCETABLE



## Appendix B

# Mutual Information of Action-Effect Pairs in Minecraft

The appendix shows the full heatmap of mutual information between actions and variables in MINECRAFT. All variables including non-effect variables are listed. Noted that although we ignore non-effect variables for further induction, the association of actions and their consequence can also be detected by calculating mutual information.

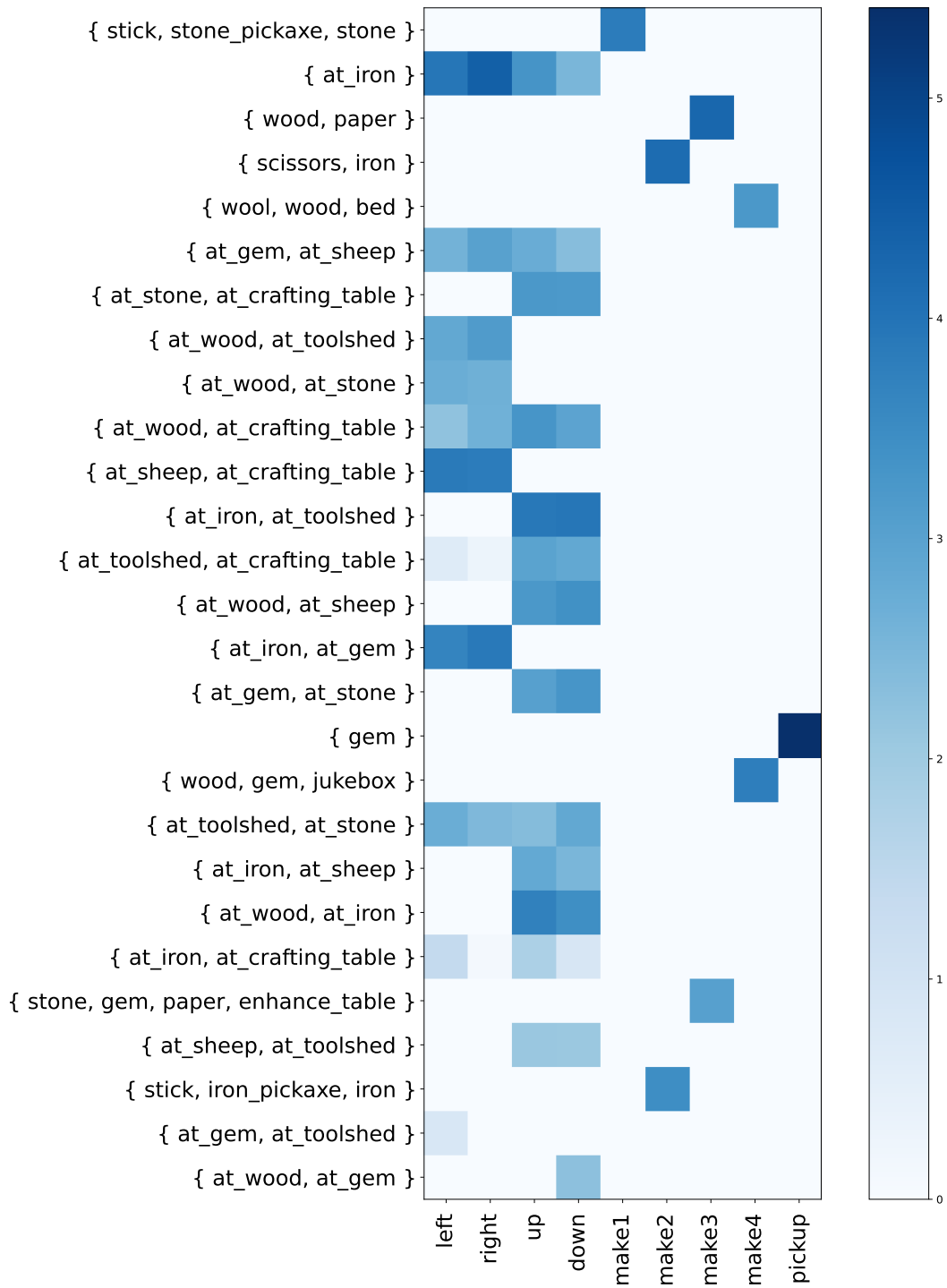
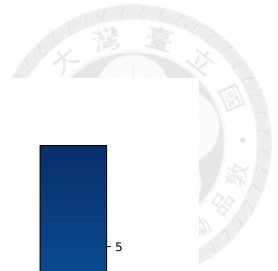


Figure B.1: Mutual information of action-effect pairs in MINECRAFT