

https://dspace.org/

Bridging Learning and Planning for Construction in a 3D environment

Muir, Nicholas

2025-01

University of the Witwatersrand, Johannesburg

https://hdl.handle.net/10539/49485

Muir, Nicholas. (2025). Bridging Learning and Planning for Construction in a 3D environment. [Master's dissertation, University of the Witwatersrand, Johannesburg]. WIReDSpace. https://hdl.handle.net/10539/49485

Downloaded from a DSpace repository. https://dspace.org/

Bridging Learning and Planning for Construction in a 3D environment

School of Computer Science & Applied Mathematics
University of the Witwatersrand

Nicholas Muir
1399185

Supervised by Dr Steven James & Prof Benjamin Rosman

January 13, 2025



A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in partial fulfilment of the requirements for the degree of Master of Science

Abstract

Building structures in a partially observable state space can be seen as a tedious and complex task for reinforcement learning. This is due to a partial observable state space only containing a fraction of the required information a reinforcement learning agent requires as well as the fact that the agent would be required to explore and perform thousands of actions to achieve anything notable. One way of reducing the complexity is using pre-learnt skills to aid the agent by decreasing the learning needed to accomplish small tasks within this domain. This, however, is still largely complex as an agent would need to select an action or skill within the pixel domain. A domain that is less complex for representing the environment and skills is that of the symbolic domain. Often used for planning, the symbolic domain contains abstracted information about the environment which decreases the complexity of the problem. Thus, we propose a framework that leverages the convenient outcome of planning methods to plan complex tasks that can be completed in any complex environment with a partially observable state space. We conduct experiments in various construction tasks in Minecraft and find that our framework outperforms a basic DQN as well as a DQN with options.

Declaration

I, Nicholas Muir, hereby declare the contents of this dissertation to be my own work. This dissertation is submitted for the degree of Master of Science in Computer Science at the University of the Witwatersrand. This work has not been submitted to any other university, or for any other degree.



13/01/2025

Contents

Preface
Abstract i
Declaration ii
Table of Contents iii

1 Introduction **1**
1.1 Sparse Reward Problem 2
1.2 Partial Observability 3
1.3 Construction 3

2 Background **6**
2.1 Reinforcement Learning 6
2.1.1 Learning in MDPs 8
2.1.2 Q-Learning 8
2.1.3 Exploration-Exploitation dilemma 9
2.1.4 Deep Q-network 9
2.1.5 Options 11
2.2 Classical Planning and Symbols 12
2.2.1 Symbolic Representation 12
2.2.2 Objects, Predicates and Effects 12
2.2.3 Planning Representations 14
2.2.4 FastDownward 16
2.3 Summary 16

3 Related Works **18**
3.1 Planning in Minecraft 18
3.2 Skills, Options and Symbolic Representation 19
3.3 Symbolic Representation with Reinforcement Learning 20
3.4 Exploration in Classical Planning and Learning 21
3.5 Classical Planning with Complex Tasks 21
3.6 Partial Observability and Uncertainty in Learning and Planning 22

4 A Proposed Framework for interleaving planning and exploration **23**
4.1 Options 24
4.2 Ground and Symbol State 25
4.3 Planning 26

4.4 Exploration . . . 26
4.4.1 Plan-Explore . . . 26

5 Applying Plan-Explore to Construction Tasks in Minecraft **30**
5.1 State Space . . . 32
5.1.1 Pixel State Space . . . 32
5.1.2 Symbolic state space . . . 33
5.2 Action Space . . . 35
5.3 Options . . . 36
5.3.1 Rewards . . . 37
5.4 PDDL Operators . . . 38
5.5 Representation of Goal . . . 39
5.6 Environment . . . 41
5.7 Experiment Setup . . . 41
5.8 Metrics . . . 42
5.9 Summary . . . 42

6 Results **44**
6.1 Sample Efficiency . . . 45
6.2 Overall Planning Time . . . 49

7 Conclusion **52**

A Planning Test Structures **54**
A.1 Line of Blocks . . . 54
A.2 Stacked Blocks . . . 54
A.3 Complex Structure . . . 55

B PDDL **56**
B.1 Planning Operators . . . 56
B.1.1 move-north . . . 56
B.1.2 move-south . . . 56
B.1.3 move-east . . . 57
B.1.4 move-west . . . 57
B.1.5 jump-north-onto-block . . . 58
B.1.6 jump-south-onto-block . . . 59
B.1.7 jump-east-onto-block . . . 59
B.1.8 jump-west-onto-block . . . 60
B.1.9 move-north-on-block . . . 61
B.1.10 move-south-on-block . . . 61
B.1.11 move-east-on-block . . . 62
B.1.12 move-west-on-block . . . 62
B.1.13 destroy-wooden-block . . . 63
B.1.14 place-wooden-block-on-floor . . . 64
B.1.15 place-wooden-block-on-block . . . 65

References **72**

Chapter 1

Introduction

Reinforcement Learning (RL) has been shown to be a powerful tool used in a wide variety of fields ranging from robot automation [Thomaz *et al.* 2005] to natural language processing [Wang *et al.* 2018]. Complex tasks may require thousands of actions to achieve a specified goal, often providing only sparse learning signals, which presents a significant challenge for reinforcement learning methods [Arjona-Medina *et al.* 2019]. Moreover, many reinforcement learning approaches make the often-unrealistic assumption of a fully observable state space, where the agent is assumed to have access to the true underlying state of the environment at each timestep. However, in real-world scenarios, only partial observations are typically available, reflecting only a portion of the true state.

For example, consider a simple grid world as shown in Figure 1.1. An agent, or player, is required to move from one room to another using a key to do so. Typically, the agent lacks access to complete information about the current state and environment, resulting in a partially observable state space. For instance, at any given moment, the agent might be located somewhere on the grid, facing a certain direction, and aware of what lies directly in front of it within a limited range. If the key is outside its field of view, the agent is unaware of the key's existence.

Comparatively, in a real-life scenario, such as a pedestrian navigating a crowded street, the agent lacks complete information about the environment For instance, the pedestrian has a goal of reaching a restaurant, but does not have full visibility of the entire city, nor does it have perfect information about the movements of other pedestrians and obstacles on the street.

These examples highlight the challenges in learning, acting, and planning in partially observable environments, contrasting simple simulated worlds like a grid world, where an agent has limited visibility and must infer hidden elements, with complex real-life situations where information is even more unpredictable and incomplete. In controlled environments such as games, where an agent may have more structured cues or accessible goals, planning can be relatively straightforward. However, in real-life scenarios, the unpredictability and partial observability of the environment significantly complicate successful planning and decision-making.

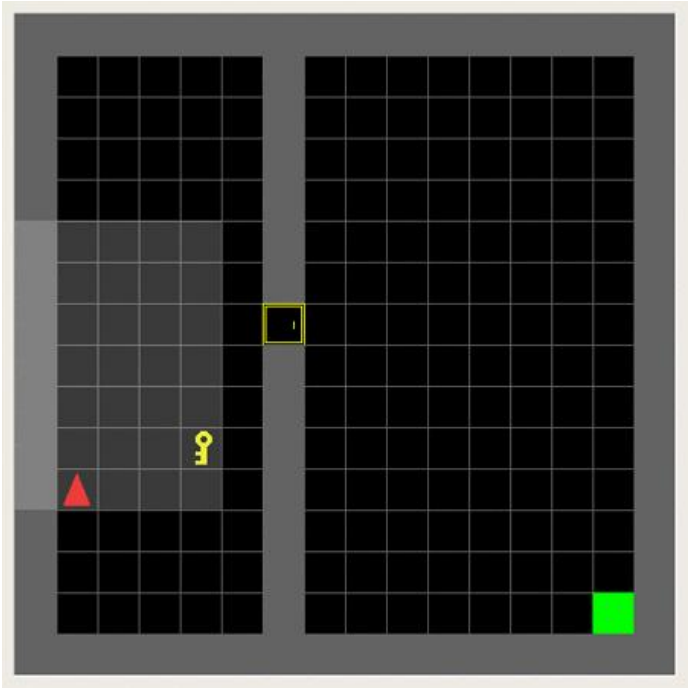


Figure 1.1: A simple grid world displaying an agent that moves from one room to the next requiring a key to move through the door [Chevalier-Boisvert *et al.* 2023]. The red triangle represents the agent within the environment, the goal is to collect the yellow key that unlocks the door and navigate to the green goal square and the light gray window represents the agents field of view.

1.1 Sparse Reward Problem

The sparse reward problem in reinforcement learning (RL) occurs when an agent receives minimal feedback from the environment, making it difficult to learn an effective policy. With limited feedback, the agent struggles to identify which actions lead to positive outcomes, especially when rewards are given only after achieving significant milestones. If the agent fails to reach these goals, it receives no reward and thus learns very little.

One approach to addressing this problem is hierarchical reinforcement learning, where options, as introduced by Sutton *et al.* [1999], act as skills or temporally extended actions. By using options, an agent can execute a sequence of actions over multiple time steps instead of choosing a single action at each step. This approach helps to

mitigate sparse rewards, as options allow the agent to pursue specific sub-goals through multi-step actions.

Classical planning provides another solution to the sparse reward problem. Here, the agent uses a model of its environment that includes the effects of its actions on the world. With this model, classical planning enables the agent to plan sequences of actions in advance, making it effective for tasks with extremely long horizons and reducing reliance on immediate rewards.

1.2 Partial Observability

While there are methods to address the sparse reward problem, partial observability compounds the challenges for both reinforcement learning (RL) and planning. In RL, a partially observable environment makes decision-making more complex, as the agent must explore unknown states and adapt to dynamic and unpredictable elements, like resource limitations and unforeseen obstacles. Since the agent has limited information, the outcomes of actions may be uncertain, leading to indecision when choosing effective actions. Partial observability can also obscure long-term dependencies, making it difficult for RL agents to learn policies that account for distant consequences of their actions. Although options help RL agents handle long-horizon tasks to some extent, partial observability still impacts their effectiveness.

Classical planning, on the other hand, often assumes a fully observable environment, where the agent has access to a complete model that includes all aspects of the environment, allowing it to plan out optimal actions over long horizons. However, creating such a model requires a human expert to manually define every detail, which can be impractical or unrealistic for complex environments. While classical planning methods are highly effective for long-horizon tasks due to their reliance on complete information, they are constrained by the need for this full model. This trade-off highlights the limitations: RL can adapt in partially observable environments but struggles with extended horizons, whereas classical planning excels in structured, fully observable settings but depends on an exhaustive environmental model.

1.3 Construction

Construction exemplifies both the sparse reward problem through only returning a reward once the structure is fully complete, and partial observability through navigation challenges as well as obscured information.

Constructing a building spans over an extended time period, resulting in an agent having to plan and make decisions over a long time horizon. RL algorithms may face difficulties in effectively managing long-term dependencies and planning sequences of actions that lead to successful goal completion.

Minecraft presents a compelling environment for RL due to several characteristics that align with the requirement for RL training, including the need for long-horizon plan-

ning Helmert *et al.* [2008] and partial observability. Minecraft is an open-world sandbox game with a vast, procedurally-generated environment. RL agents trained in Minecraft can learn to navigate, interact and manipulate the environment. These environments can be partially observable as various obstacles could hinder the agents visibility. One challenge posed by Minecraft is that tasks require an agent to chain thousands of low-level actions together to achieve anything meaningful. One approach to mitigate this problem is with the use of options, where options circumvent the need for employing low-level actions by providing higher-level action sequences to accomplish tasks. Competitions around constructing buildings and settlements within Minecraft have arisen due to the complexity and nature of this problem. One competition that covers and promotes this field is the MineRL competition [Shah *et al.* 2021]. The MineRL competition serves as a benchmark for RL agents to learn skills that can be achieved in the real world. The final evaluation is based on human evaluation which allows for a higher degree of accuracy. However, the majority of these approaches Use behavior cloning [Hussein *et al.* 2017] to preemptively learn sub-tasks that facilitate the learning process by leveraging existing demonstrations.

Figure 1.2 illustrates the complexity an agent faces when building even a basic house in a complex environment. Without a defined plan, the agent has countless possible actions to accomplish a single sub-task, not to mention constructing an entire house. This presents a significant challenge that a reinforcement learning agent must overcome.

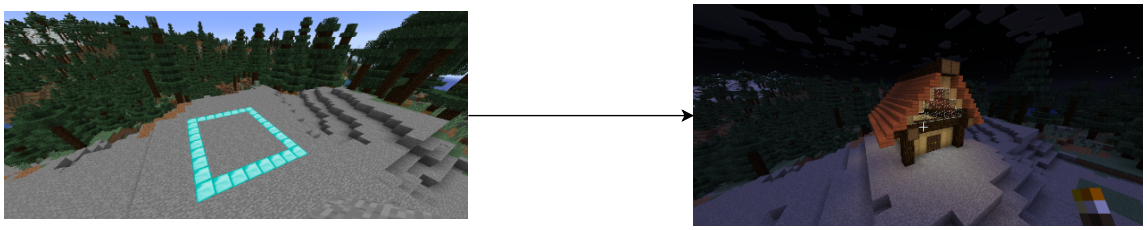


Figure 1.2: Building a house leaves many questions including where to start and how to start. This shows how complex a feat it is to construct a structure with no prior knowledge. To construct this house, a total of 178 blocks are needed which requires collecting each block from the surrounding area as well as crafting blocks into different forms. The initial image displays an initial outline of where the house should be constructed. The second image displays the goal of what the house would eventually look like given the resources are available to build the example house.

Since options aid RL with long horizon tasks, it seems a logical step to leverage options to aid an RL agent in the domain of Minecraft in completing more complex tasks without needing to deal with chaining primitive actions, such as, moving to travel long distances. Even with options available to an agent, this class of problem is still complex as we have to reason, learn, explore and plan in the pixel space. Conversely, classical planning approaches, exemplified by Fast Downward [Helmert 2006], have demonstrated remarkable efficacy in tackling complex tasks that may seem otherwise impossible for RL. Though promising, they require that the agent has a fully observable state space to achieve the given goal, as well as requiring symbols to be predefined.

To use conventional planners such as Fast Downward, we use symbols to represent the different elements of the planning domain. Since Minecraft provides states that retain pixel domain characteristics, we would need to convert these observations into a symbolic format. Symbols are abstracted entities that represent aspects of the planning domain. These aspects are the states, actions, variables, and goals that together make up the symbolic domain.

While classical planning methods are desirable for their efficiency, they require a model of the agent's behaviors. This model can either be predefined or learned, as seen in the work by [Konidaris *et al.* 2018]. However, this still leaves the challenge of specifying the full domain, including all objects and their relationships. In environments like Minecraft, where states are represented in a pixel-based format, defining this full domain can be a significant challenge.

To address this, we propose a framework called Plan-Explore, which leverages the benefits of classical planning while overcoming the limitations of requiring a fully observable state space. This framework enables goal achievement in complex, partially observable environments and bridges the gap between symbolic and pixel-based domains.

In this dissertation we begin with the background in the fundamentals and specific methods in Chapter 2 such as RL and symbolic representation that are used in this research. We then move onto related works Chapter 3 which comment on the different fields of research all relevant to the techniques used in this research from different planning techniques used, to how different options are represented and learnt. Chapter 4 discusses the details of our framework, and how it interleaves planning and exploration. We discuss in detail each aspect of this framework and how each section contributes to the overall performance in achieving tasks in a partially observable environment. We then discuss the experiments in Chapter 5 where our framework is implemented into the partially observable environment of Minecraft to achieve complex tasks such as building different structures. We then discuss these experiment results in Chapter 6 where we highlight the differences in environment interactions between each model used. Lastly, we conclude with our findings in Chapter 7 and highlight the contributions our findings make to the future of planning.

Chapter 2

Background

In this chapter, we outline relevant background information and theory required for the methods that will be used. We begin with Section 2.1 where we define reinforcement learning and the foundational principles and techniques that characterize this dynamic field. Through a detailed examination of the agent-environment interaction loop shown in Figure 2.1, reward maximization and key elements such as states, actions and polices, we establish a conceptual framework that forms the basis of RL. After which, we focus our view deeper into specific aspects of RL, namely the Markov Decision Process in Subsection 2.1.1, Q-learning in Subsection 2.1.2, and Deep Q-Networks in Subsection 2.1.4. Finally, we explore options in Subsection 2.1.5 where we highlight how options offer a hierarchical structure for action selection, allowing for efficient learning and decision-making in complex environments. Having explored the dynamic realm of RL, we then pivot towards classical planning methods in Section 2.2 wherein we discuss a contrasting approach to problem solving. We first highlight classical planning as a deterministic problem-solving approach. We then delve into the basic concepts of classical planning and explore the importance of planning representations in Subsection 2.2.3. Then we discuss Symbolic Representations in Subsection 2.2.1 and highlight the advantages of symbolic reasoning, such as compact representation and reasoning efficiency.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is seen as an optimization framework in which an agent interacts with an environment in an attempt to complete a given task in a trial-and-error fashion, where each action an agent takes within an environment returns a state and a reward.

The agent perceives the environment through observations. These observations provide information about the current state of the environment which an agent will use to make decisions. The agent interacts with the environment by carrying out an action which is based off of the observed state. This action can be discrete or continuous.

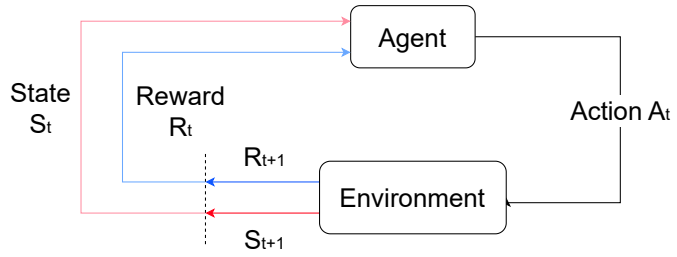


Figure 2.1: Reinforcement learning training loop displaying the agent-environment interaction [Sutton and Barto 2018]

The Markov Decision Process (MDP) is a foundational framework for decision-making under uncertainty. Typically, an environment is modelled as a Markov decision process $\langle S, A, P, R, \gamma \rangle$, where (i) S is the state space; (ii) A is the set of actions available to an agent; (iii) $P(s'|s, a)$ is the transition dynamics, specifying the probability of an agent finding itself in state s' after executing action a from state s; (iv) $R(s, a)$ is the reward function that specifies the reward an agent receives for executing action a in state s; and (iv) $\gamma \in [0, 1)$ is used to discount future rewards.

Within this agent-environment interaction loop represented in Figure 2.1, several components serve an important role in facilitating the learning process. At each time-step t an agent is in a state s_t where $s_t \in S$, and S is the set of all possible states in the environment. The agent can take an action a where $a \in A$ and A is every possible action the agent can take within the environment. When the agent takes action a_t on the environment, the environment returns a reward r_{t+1} and new state s_{t+1} where $t + 1$ represents the subsequent time step after the action a_t has been taken. The reward signal r_{t+1} where $r_{t+1} \in \mathbb{R}$ refers to the reward returned from the environment for achieving the next state s_{t+1} from s_t. After the agent receives r_{t+1} and s_{t+1}, it updates its knowledge by incorporating the new experiences. s_{t+1} replaces s_t as the new current state the agent is in for the next decision-making step. This loop continues to make decisions in order to maximize the total cumulative reward over time [Sutton and Barto 2018].

The environment is the world the agent resides in and as the agent takes actions within this environment the environment transitions to a new state. This transition is due to the dynamics of the environment which represents the laws that govern how the environment transitions from one state to the next. The transition dynamic of an environment is typically represented as a transition probability $P(s', r \mid s, a)$, which shows the probability of transitioning to state s' with received reward r when taking an action a in state s.

$$P(s', r \mid s, a) = \Pr\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\} \tag{2.1}$$

The transition probabilities displayed in Equation 2.1 capture the probabilistic nature of the environment, this being how likely an agent is to move from one state to another gives the action taken in the current state.

2.1.1 Learning in MDPs

An agent interacts with its environment through a *policy* π, which maps states to actions. The utility of a given policy π can be quantified by its *value function*, which captures the expected future rewards following π. The aim of an agent is to discover an optimal policy π^*, such that $\pi^* = \underset{\pi}{\text{argmax}}\, v_\pi(s)$ for all s in S.

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s \right] \tag{2.2}$$

When evaluating policies we use what is known as a value function. Value functions are valuable tools in RL that evaluate the quality of states and state-action pairs. We formally define the state value function (V-function) represented in Equation 2.2, as the expected return from state s following the policy π. The return from this equation is the sum of the discounted rewards received over an infinite time horizon [Sutton and Barto 2018]. $v_\pi(s)$ represents the expected return of state s under the policy π, $\mathbb{E}_\pi$ is the expectation operator under π and represents an expectation that this value comes from all possible sequences of states and actions determined by policy π. Gamma, γ, represents the discount factor and influences the importance of future rewards compared to immediate rewards. Lastly, $r(s_t, a_t)$ is the expected reward when taking action a_t in state s_t at time step t.

When the transition dynamics T and reward function R are known, we use planning such as policy iteration [Howard 1960], otherwise a model-free algorithm such as Q-learning is required [Watkins 1989].

2.1.2 Q-Learning

Q-learning is a model-free, off-policy, temporal difference control algorithm used in finding optimal action-selection policies for MDPs [Watkins 1989]. Q-learning uses the action-value function Q which approximates the optimal action-value function q_*. This being that the next best action a is selected based off of a given state s. Thus, for each step on an episode the action value function Q is updated as follows :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.3}$$

The Q-function (action-value function) is denoted by $Q(s_t, a_t)$ which is the current estimate of the Q-value, where the current state and action are denoted by s_t and a_t respectively . α represents the learning rate which determines to what extent the new information will override the current Q-value estimates. γ represents the discount factor and influences the importance of the future reward relative to the current reward. $t+1$ represents the next time step and thus r_{t+1} denotes the reward received at time step $t+1$. s_{t+1} represents the state the agent will be in after action a_t is taken in state s_t. $\max_a Q(s_{t+1}, a)$ is the maximum expected future reward from all possible actions given the next possible state s', and represents the maximum expected reward achievable from state s' onwards.

2.1.3 Exploration-Exploitation dilemma

One question that arises is how an agent should act in its environment to collect samples for performing the update in Equation 2.3. This is known as the exploration-exploitation dilemma, a fundamental concept in RL. It represents the challenge faced by agents when deciding whether to explore unknown actions to gather more information or to exploit already known actions to maximize the immediate rewards. The trade-offs between exploration and exploitation hold large significance as to how the agent performs in a given environment. When an agent favours exploration, the agent tends to use new actions and strategies to gather more information about the environment and the different rewards associated with these actions. Exploration in terms of Q-learning is essential to find better actions that may lead to higher rewards in the long term. If exploration is not sufficiently used, the agent can miss information of the environment and results in a sub-optimal policy.

On the other hand, exploitation involves selecting already known actions according to the rewards that would be received based on the current estimates calculated by the Q-function $Q(s_t, a_t)$. Exploitation is used by the agent to maximise the immediate rewards by using the current knowledge of the environment. Exploitation is vital for Q-learning to make decisions that will lead to high rewards based on the current knowledge of the environment.

In RL, it is vital to balance exploration and exploitation to achieve efficient learning of the environment. One such strategy in handling this balance is the ϵ-greedy strategy. The ϵ-greedy strategy influences the agent to select an action based off of the following policy. With a probability of $1 - \epsilon$ where $0 \leq \epsilon \leq 1$, exploitation occurs and the agent selects an action a in the current state s that will return the highest value from the Q-function. With a probability of ϵ, exploration occurs, and random actions from the action space A are selected.

2.1.4 Deep Q-network

In order for Q-learning to converge to an optimal solution, the agent must explore all possible states in the environment. Deep Q-networks however mitigate this need by leveraging function approximation to estimate the Q-values for unseen states. The Deep Q-network (DQN) [Mnih *et al.* 2015] is an algorithm that extends Q-learning to higher dimensions with the use of Neural Networks (NN). By using function approximation through the use of these neural networks, DQNs are capable of approximating the Q-values for a wide range of states, allowing DQNs to handle environments with a large state space.

A widely used approach in deep learning and gaming is the DQN algorithm, which learns through trial and error, mimicking human-like learning. Since DQN is a form of Q-learning, it uses a modified version of the Q-learning update, but with a neural network to approximate the Q-values. The algorithm uses a target network to mitigate slow convergence and instability in training. The target network is a duplicate of the main Q-network, updated periodically rather than at each step. The network has

parameters denoted by θ^{pred}, whereas the target network has parameters denoted by θ^{target}.

This stabilization is achieved through the DQN loss function:

$$L(\theta) = \left(r + \gamma \max_{a'} Q(s', a'; \theta^{target}) - Q(s, a; \theta^{pred})\right)^2 \tag{2.4}$$

The DQN algorithm uses a target network to improve convergence and stability during training. The target network, which is a duplicate of the main Q-network, is updated periodically by copying the parameters from the main network. This periodic update stabilizes learning by ensuring that the loss function is calculated using Q-values from the more stable target network. The main network's parameters are then updated based on this loss using an optimization algorithm, allowing for more reliable training.

Algorithm 1 Deep Q-Network (DQN) with Replay Buffer

Initialize:
  Q-network with random weights θ
  Target Q-network with weights θ_{target}
  Replay buffer $\mathcal{D}$ with capacity N
  Learning rate α, discount factor γ, exploration rate ϵ
  Batch size B
for each episode **do**
  Initialize state s_0
  for each timestep t in the episode **do**
    if random ¡ ϵ **then**
      Select action a_t randomly (exploration)
    else
      Select action $a_t = \arg\max_a Q(s_t, a; \theta)$ (exploitation)
    Take action a_t, observe reward r_t and next state s_{t+1}
    Add transition (s_t, a_t, r_t, s_{t+1}) to replay buffer $\mathcal{D}$
    if learning step threshold reached **then**
      Sample a batch B_{sample} of B transitions (s, a, r, s') from $\mathcal{D}$
      for each transition (s, a, r, s') in the batch **do**
        Compute target: $y = r + \gamma \max_{a'} Q(s', a'; \theta_{\text{target}})$
        Compute loss: $L(\theta) = (Q(s, a; \theta) - y)^2$
      Perform gradient descent to minimize $L(\theta)$ and update θ
    if every C steps **then**
      Update target network: $\theta_{\text{target}} \leftarrow \theta$

The DQN algorithm often uses a replay buffer, as described by Mnih *et al.* [2015], as seen in algorithm 1. Replay buffers are data structures used to store experience. This experience often takes the form of (s,a,r,s'), this being that state information is stored such that when recalling the experience we have the current state s, action taken a, the reward that was received r and the next state s' that was a result of taking action a in state s. The use of a replay buffer is to prevent highly correlated samples which slows

convergence and decreases stability within the DQN. This is achieved by randomly sampling previous experience from the replay buffer to update the model, this allows the agent to learn from a variety of experience improving generalisation.

Even when an environment is partially observable, the DQN algorithm is still a popular choice in solving tasks. This can be seen through Frazier and Riedl [2019] showing that variations of the DQN algorithm successfully achieve the task of navigating a maze and reaching the goal location.

2.1.5 Options

Options are temporally-extended actions that can be seen as sub-policies, enabling agents to perform sequences of actions to achieve goals [Sutton *et al.* 1999]. They provide a level of abstraction when performing tasks by allowing the agent to execute complex behaviors over multiple time steps, thus eliminating the need to explicitly consider each individual action. Using options allows an agent to focus on learning and executing temporally extended behaviors rather than learning to chain large sequences of primitive actions. This temporal abstraction makes decision-making more efficient.

The options framework formally described in Sutton *et al.* [1999] defines an option $o \in O$ as:

$$o = (I_o, \pi_o, \beta)$$

where:

- π_o is a policy that, when followed, selects actions until the option terminates.
- $\beta : S \rightarrow [0, 1]$ is the termination condition, which gives the probability of the option terminating in a given state.
- I_o is the initiation set, a subset of states $I_o \subseteq S$ where the option can be initiated.

An option can be executed if the current state $s \in I_o$, and it operates in a call-return manner. When an agent selects an action a using the policy π_o in state s, the environment transitions to state s'. If s' satisfies the termination condition $\beta(s') = 1$, the option terminates, signaling that the goal or subgoal associated with the option has been reached. If s' does not satisfy the termination condition, the option continues using the policy π_o until termination. Actions can be seen as a special case of options, where the set of options, I, is equal to the set of states, S. In this case, the policy selects an action for each state, and the action terminates after one step, without requiring further decisions. This is a simplified form of an option, where each action is an immediate, one-step choice rather than a sequence of actions.

Options can be both predefined or learnt. Predefined options are manually designed by a human expert based on the knowledge of the domain.

Using Options with DQNs and Q-learning

In Deep Q-Networks (DQNs) and Q-learning, options are used as temporally extended actions that help structure an agent's decision-making process across multiple time

steps. By introducing options, the agent can group a series of actions into a higher-level strategy aimed at achieving subgoals.

In these algorithms, the agent typically learns a Q-value function over state-action pairs. When options are introduced, the agent extends this function to learn over state-option pairs. This allows the agent to select an option that executes a series of actions (guided by a sub-policy) until it terminates. Instead of learning the value of each individual action, the agent learns the value of entire options at each state, enabling more effective long-term planning. The learning rule remains largely the same, with the primary difference being that the agent learns the value of options, rather than actions, at each state.

2.2 Classical Planning and Symbols

Classical planning stands as one of the cornerstones of artificial intelligence [Russell and Norvig 2016]. Classical planning represents a problem in an abstracted manner ignoring all irrelevant details and finds sequential actions that start at an initial state, and end at a goal state in deterministic environments. When viewing an environment for classical planning, the set of states extracted from the environment hold select information known as predicates which are the properties and logical statements about the objects in the environment. Actions that can be taken in the environment, transition the environment between states. The action's preconditions dictate when the action can take place, and once the action takes place, the effects update the environment state.

Classical planning is powerful. but requires a human to specify the environment. Because of this, recent work has focused on automatically learning these representations from data

Given that planning requires predicates, preconditions and effects, a common standard to use is the Planning Domain Definition Language (PDDL).

2.2.1 Symbolic Representation

PDDL operators often viewed as actions closely resemble that of Options defined in Section 2.1.5. A PDDL representation is a depiction of the physics of the domain in a symbolic form. This ranges from the possible actions that can be taken within the environment to the effects the actions have on the environment [Aeronautiques *et al.* 1998].

2.2.2 Objects, Predicates and Effects

The PDDL representation often assumes the world consists of objects, where these objects represent this environment in a symbolic manner. Objects can range from representations of entities within the world to specific instances. Using Minecraft as an overarching example, we can view objects as the different blocks scattered in the environment, different items the agent can hold, and the agents that exist in the world.

```
(:objects
    block1 block2 block3 - block
    stick - item
    agent - steve)
```

Predicates are the properties or logical statements about the objects within the world.

An example of a predicate for the Minecraft domain would be whether the agent is currently holding an item or not. This would be shown below where `x` denotes the agent's hand. `EMPTY` shows true if `x` is not currently holding an item, `AXE` will show true if `x` currently holds an Axe.

```
( :predicates (EMPTY ?x) (AXE ?x) )
```

Thus these objects and predicates are the symbols that represent the current state of the environment, and are supplied to the agent before planning commences.

Preconditions

Preconditions are the prerequisites that are required before an action can be executed. Using another example of a Minecraft operator we can view the preconditions of placing of a block on the floor as follows :

```
:precondition (and
    (are-seq ?y_start ?y_end)
    (are-seq ?n_end ?n_start)
    (agent-has-n-wooden-blocks ?ag ?n_start)
    (at-x ?ag ?x)
    (at-y ?ag ?y_start)
    (at-z ?ag ?z)
    (not (block-present ?b))
)
```

This precondition code can be read as so, are-seq statements check if the two values are sequential. This meaning that `y_start` and `y_end` are sequential values as well as the pair `n_end` and `n_start`. The code section `agent-has-n-wooden-blocks` defines a boolean value as to whether the agent `ag` and the number of wooden blocks `n_start`. Next are the location values of the agent defining the three dimension `x,y,z` and lastly the condition that there is no block `b` present in front of the agent.

If all these preconditions have been found to be true then the option of placing a block on the floor is viable when planning in the current state.

Effect

The effect described within an action describes what states should be updated within the symbolically represented world. Viewing the example below:

```
:effect (and
    (not (at-z ?ag ?z_start))
    (at-z ?ag ?z_end)
)
```

This example is of the effect an agent would have while moving north. This can be seen through the update to the value `at-z`, by defining that the agent is `not` as `z_start` but rather at `z_end`.

2.2.3 Planning Representations

In the given state space S, the agent can assess specific states within the environment, denoted by $s \in S$. The predicates of a system represent the properties and relationships within a given state and are used to describe the current configuration of the environment. Once assessed the agent can select a set of action operators (operators) $a* = a_1, a_2, ..., a_n \in A$ where A is the set of all possible actions in the environment, after which the agent will then use the set of possible actions $a*$ to create a sequence

of actions which we will formally call a plan p, when sequencing the agent will take into account all preconditions and effects of each action to find the best suitable plan for achieving the goal. Once a viable plan has been found the agent is able to execute it to achieve the goal defined [Aeronautiques *et al.* 1998].

PDDL operator

Operators are the individual actions that can be used within a PDDL. These PDDL operators can be seen as analogous to options in RL. A representation of an action in a PDDL can be seen below where the action of an agent moving north is described.

```
(:action move-north
    :parameters (?ag - agent ?x - position ?y - position ?
       z_start - position ?z_end - position)
    :precondition (and
        (agent-alive ?ag)
        (agent-on-floor ?ag)
        (at-x ?ag ?x)
        (at-y ?ag ?y)
        (at-z ?ag ?z_start)
        (are-seq ?z_start ?z_end)
        (not (exists (?bl - block) (and
            (block-present ?bl)
            (at-x ?bl ?x)
            (at-y ?bl ?y)
            (at-z ?bl ?z_end)
        )))
    )
    :effect (and
        (not (at-z ?ag ?z_start))
        (at-z ?ag ?z_end)
    )
)
```

This action uses the parameters of the agent's current location and the end z value coupled with preconditions that the agent is alive, is on the ground, the current agent location is at the coordinates *(x, y, z_start)*, the coordinates *z_start* and *z_end* are sequential and that no block exists north of the agent. If this precondition is met the action can take place, wherein the effect this action has updates the agent's location from *(x, y, z_start)* to *(x, y, z_end)*.

PDDL goal

In PDDL, the goal state defines the specific conditions that must be achieved for a plan to be successful. Referring to the example effect given in subsection 2.2.2, we can express this goal condition as follows:

```
(:goal (at-z ?ag ?z_end))
```

In this case, the goal condition would ensure that the agent, represented by `?ag`, has reached the specific `z_end` position and is no longer at the `z_start` position. By setting these conditions, PDDL guides the planner to generate actions that achieve the goal while considering the environment's constraints and dynamics.

2.2.4 FastDownward

FastDownward is a well-established planning system developed for classical planning problems. Introduced in Helmert [2006], FastDownward is a widely used planner. One of its key innovations is the ability to translate propositional planning tasks encoded in PDDL (Planning Domain Definition Language) into multi-valued representations, which allow for a more structured exploration of the search space. Fast Downward comes with many search algorithms and heuristics, allowing it to handle a variety of planning tasks across different domains. These include classical algorithms like A*, which finds optimal solutions based on a heuristic cost estimate, and Enforced Hill-Climbing (EHC), which incrementally searches towards lower-cost states, escaping local optima through breadth-first exploration. It also offers greedy algorithms like Lazy Greedy and Epsilon-Greedy, which balance efficiency and exploration by prioritizing high-potential paths based on heuristic values.

In terms of heuristics, Fast Downward includes options like the Fast Forward (FF) heuristic, which approximates goal cost by ignoring negative effects; Additive (Add), which sums sub-goal costs independently; Causal Graph (CG), which evaluates dependencies between variables; and Blind, which assigns equal cost to all states. Each combination of algorithms and heuristics is tailored to suit specific planning scenarios, making Fast Downward a versatile tool for handling complex state spaces.

2.3 Summary

This chapter provides an in-depth exploration of the foundational methods and theoretical concepts that underpin this research, encompassing reinforcement learning (RL) fundamentals, planning approaches, and options-based abstractions. We begin with RL basics, detailing the training loop and transition probabilities used to model an environment's uncertainty. We then introduce the Markov Decision Process (MDP), a formalism for decision-making under uncertainty, and discuss value functions, which help assess the quality of states and actions. Depending on the availability of transition probabilities and rewards, either planning or Q-learning can be employed to find an optimal policy.

Q-learning, a temporal-difference method, optimizes action values by maximizing cumulative rewards across state-action pairs. We discuss the balance between exploration and exploitation, including the ϵ-greedy exploration strategy, and explain how this impacts effective learning. Next, we introduce the Deep Q-Network (DQN) algorithm, noting its use of neural networks to minimize temporal-difference errors and the replay buffer's role in training stability and efficiency. In more complex environments, options enable the abstraction of primitive actions into high-level sub-policies, which aids in efficient long-term planning and makes DQN with options a suitable benchmark for partially observable tasks.

We also address classical planning, designed for fully observable, deterministic environments with discrete actions. Here, the Planning Domain Definition Language (PDDL) is highlighted for defining predicates, preconditions, and effects. The chapter concludes with a transition to the following section, which reviews related work in planning, options, symbolic RL integration, and task-solving approaches in domains like Minecraft.

Chapter 3

Related Works

This chapter presents an overview of relevant literature and frameworks across multiple domains pertinent to this study. Section 3.1 discusses planning methods implemented in the Minecraft environment. In Section 3.2, we examine methods for learning options in both learning and planning contexts. Section 3.3 explores approaches that integrate symbolic representation with reinforcement learning. Next, Section **??** reviews exploration strategies in both classical planning and learning. Section 3.5 addresses works in classical planning that aim to solve complex tasks, and finally, Section 3.6 covers research on addressing partial observability in both learning and planning.

3.1 Planning in Minecraft

Many works make use of the domain of Minecraft in the planning and assembling of structures. Wichlacz *et al.* [2019] uses a variety of planning models including a classical planning model, as well as hierarchical planning models, PANDA [Bercher *et al.* 2014] and SHOP2 [Nau *et al.* 2003]. The initial classical planning model used contained the two primitive actions of placing and removing blocks. These actions used a 3D coordinate grid in order to navigate and act at the location specified, but as stated, this simple model ignores the high-level structure and makes a high-level structure virtually impossible to construct. To overcome this limitation, auxiliary actions are introduced to aid in the construction of high-order structures. These auxiliary actions contain extra information necessary for construction that would not normally appear in primitive actions.

Hierarchical Planning Models use defined high order tasks that are then decomposed down into primitive actions. Two such models are the PANDA and SHOP2 Models that follow similar methods of decomposing the higher order tasks into primitive actions. Outcomes from Wichlacz *et al.* [2019] show that it is possible for planners to plan the assembly of a structure with PANDA and SHOP2 models being more successful than others. Learning and representing these primitive actions require adaptions to that of a standard PDDL.

Construction in a 3D environment is a popular topic with many competitions revolving around it. MineRL doubles as both a competition and a platform that provides a large scale dataset of human demonstrations that can be used to train an RL agent in skills to accomplish tasks in a Minecraft setting [Guss *et al.* 2019]. This competition creates challenging tasks that are complex to hard-code and thus require new techniques to accomplish [Guss *et al.* 2020]. Other attempts at solving Minecraft tasks such as Aluru *et al.* [2015] show how the A* search algorithm can be implemented in Minecraft through converting Minecraft problems into Object-Orientated MDPs. This is done through only specifying the relevant object needed to represent the current world. Benyamin *et al.* [2023] proposes a method in which RL is used to learn domain models through observations. Although this is shown to be successful, an assumption is made that the agent is in a fully observable state space.

3.2 Skills, Options and Symbolic Representation

Learning options is usually achieved within the state space of the environment. This poses difficulty when the aim is to transfer learnt options to new tasks an agent needs to complete. Konidaris and Barto [2007] demonstrate a method in which options are learnt in the portable, egocentric observation space rather than that of the state space, thus allowing for easier portability of these learnt options into unseen tasks. Using options to learn a symbolic representation is another hurdle. This of course can be achieved manually but would be time-consuming and would require a certain level of expertise in the field. Konidaris *et al.* [2018] mitigate these issues and demonstrate a method in which an agent is able to learn these symbolic representations. These representations are learned by collecting data from the agent's interactions with the environment, including transition data generated as the agent executes an option and gathers the resulting information.

Other approaches, such as [Ugur and Piater 2015a], focus on generating symbolic representations directly from sensory data and interactions with the environment, improving these symbols based on the experience gained. While this system allows for planning to take place on a real robot, it requires objects to be specified before the learning process begins. However, a key limitation is that there is a need for pre-defined features that guide the discovery process. In this light, [Ugur and Piater 2015b] emphasizes refining symbolic representations through multi-step interactions, allowing agents to iteratively adjust symbols as they gather more experience. However, refining symbols over multiple steps may introduce inconsistencies in highly dynamic environments, which can lead to symbols that are only partially relevant or temporarily accurate.

Another approach is that of Christoffersen *et al.* [2023], which focuses on creating symbolic representations for RL in non-Markovian environments. This is achieved through the proposed algorithm AutRL, which leverages learned symbolic representations to guide RL agents. Despite its advantages, one major constraint is that it relies on Deterministic Finite Automata (DFA) for representing symbolic knowledge, which means it is limited to environments where the underlying decision process can be represented as a regular language. This constraint limits its applicability in more complex environments,

like Minecraft, where the dynamics are highly irregular and characterized by intricate, real-time interactions.

Learning portable representations allows for transfer of options between different environments. James *et al.* [2020b] highlight learning portable representations for agents with differing environments. With this approach, the number of tasks required to reach a goal significantly decreased the number of tasks the agent was required to learn. Similarly James *et al.* [2020a] learn to accomplish specified tasks and transfer these learnt actions into an object-centric symbolic representation. These representations are then tested for portability between tasks with similar objectives, and are proven to aid an agent in completing new tasks and decreasing the overall number of operations required for an agent to complete these tasks. This demonstrates that learned symbolic representations of options can be both acquired and transferred across different environments.

3.3 Symbolic Representation with Reinforcement Learning

Many works have favoured combining symbolic representation with that of RL, such as using planning to aid a RL agent to complete complex tasks within a 2D domain.

Frameworks such as PEORL [Yang *et al.* 2018] and SDRL [Lyu *et al.* 2019] accomplish this task in different manners. PEORL integrates symbolic planning with hierarchical RL to assist an agent in decision-making in changing environments. This framework uses symbolically-represented actions and constraints to create a symbolic plan. This plan maps to a sequence of options that a RL agent can leverage. Since RL can be optimized over its iterations, this framework is able to optimize the symbolic plan with updated mappings to options. The testing domains used are the 2D environments of the Taxi-Domain [Barto and Mahadevan 2003] and Grid-world [Leonetti *et al.* 2016]. The overall outcome of this framework was the simultaneous improvement to both symbolic planning and RL.

SDRL, on the other hand, integrates symbolic representation into deep reinforcement learning to accomplish the task of handling high-dimensional inputs and symbolic planning. The composition of this framework uses symbolic knowledge of the domain to plan sequences of actions in order to achieve a goal, using deep reinforcement learning to learn policies for the symbolic actions and proposes new goals based off of the training performance of the agent.

Both PEORL and SDRL use symbolic planning with RL to achieve complex tasks within the 2D domain. The question remains whether an approach using partially observable symbolic representations coupled with RL can achieve goals within a 3D environment.

Illanes *et al.* [2020] explores the use of symbolically-represented actions to define sets of goals to generate rewards that will correspond with these actions. Achieving a taskable agent with tasks defined by a human within the symbolic domain as well as improving the efficiency of sample generation using plans from previously learnt data.

3.4 Exploration in Classical Planning and Learning

Exploration plays a crucial role in both classical planning and reinforcement learning (RL) when dealing with complex environments. In FastDownward [Helmert 2006], search algorithms traverse the state space to reach the goal state, while in RL, strategies such as Go-Explore [Ecoffet *et al.* 2019] tackle exploration in high-dimensional or sparse-reward environments by revisiting and exploiting previously explored states.

Go-Explore addresses hard-exploration problems by creating an archive of explored states, enabling the agent to expand its knowledge by revisiting them. This method significantly improves exploration in environments where rewards are sparse, and traditional exploration techniques fail. However, Go-Explore faces challenges in dynamic and unpredictable environments. Its reliance on stable states to revisit limits its effectiveness in settings where the environment changes over time or in response to the agent's actions.

Pathak *et al.* [2017] introduces an intrinsic curiosity reward that drives exploration based on prediction errors. In this method, the agent is rewarded for discovering novel states by attempting to predict the outcomes of its actions; any significant prediction error signals a novel state, encouraging the agent to explore it further. This curiosity-driven approach is particularly useful in sparse-reward settings, where external rewards are rare, as it continually motivates the agent to discover and interact with previously unseen parts of the environment.

Exploration in classical planning can also be achieved through approaches like those proposed by Lipovetzky and Geffner [2017], which strikes a balance between exploration and exploitation by prioritizing states based on both heuristic values and "novelty." This method ensures that new or unique areas of the state space are explored, enhancing the search's breadth and reducing the risk of getting trapped in local optima. However, a limitation of this approach is that tracking and evaluating novelty can become computationally intensive, especially in highly dynamic or large environments.

3.5 Classical Planning with Complex Tasks

Solving complex tasks has been a hot topic within the world of classical planning with many attempts and strategies to further the capabilities of planning. One such strategy is to address the scalability issues when planning. Helmert [2006] uses advanced heuristic search techniques to improve the efficiency and scalability of classical planning, allowing it to handle larger state spaces and more complex tasks. However, Fast Downward struggles when not all information is known. Since classical planning assumes a complete and fully defined state space, if a symbol or piece of information is missing or unknown, Fast Downward cannot effectively search or use it. This limitation may prevent the planner from reaching the goal, as it cannot account for all potential actions or states that depend on that missing information. Another approach in furthering planning in complex domains is Kambhampati and Srivastava [1996], where the combination of state and plan space refinement strategies allows for more flexible and efficient planning. However, despite its advantages in certain contexts, this ap-

proach still struggles in domains that are not fully observable, as it assumes complete knowledge of the environment, limiting its effectiveness in real-world applications with partial observability or uncertainty.

3.6 Partial Observability and Uncertainty in Learning and Planning

Due to classical planning approaches assuming complete knowledge of the environment, they struggle when applied to real-world scenarios that are only partially observable. In such settings, planners may lack critical information about certain states or actions, limiting their effectiveness and often resulting in incomplete or suboptimal plans. Probabilistic reasoning [Boutilier *et al.* 1999] is a concept developed for reasoning in learning and planning, particularly when dealing with uncertainty in the environment. This is achieved by assigning probabilities to states or outcomes allowing planners the ability to consider multiple possible states when planning even with missing or uncertain information. While probabilistic reasoning provides a framework for handling uncertainty in planning, it does not fully address the challenges posed by environments where certain states remain unobservable or hidden from the agent.

To address these limitations, Partially Observable Markov Decision Processes (POMDPs) [Kaelbling *et al.* 1998] provide a framework for decision-making in environments where not all states are observable. This is achieved by maintaining a belief state, which probabilistically represents the possible states of the environment based on the available observations. Although POMDPs offer a valid approach for exploration in partially observable environments, they encounter significant difficulties in complex settings. The key challenge is state space explosion, where the belief space grows exponentially as the environment's complexity increases. This limits the practical effectiveness of POMDPs in real-world scenarios with large, dynamic state spaces [Thrun 2000].

Chapter 4

A Proposed Framework for interleaving planning and exploration

As discussed in the background [Section 2.1] reinforcement learning trains through interactions within an environment to achieve a goal. An RL agent learns through trial and error, allowing it to explore environments and gather new information to achieve its tasks. This trial-and-error approach requires extensive interactions with the environment, as the agent incrementally discovers and refines optimal strategies. As a result, the learning process can be highly time-intensive. Many problems addressed by RL involve high-dimensional states and actions which exacerbate the time and number of environment interactions it would take to find an optimal policy.

Classical planning, as defined in Section 2.2, can serve as an option to mitigate some limitations inherent in RL. However, for classical planning to be effective, it generally requires a fully observable state space—a rare luxury in the complex environments often encountered today. While methods designed for partially observable domains, such as POMDP-based approaches [Kaelbling *et al.* 1998] like POMCP [Silver and Veness 2010] and Point-Based Value Iteration [Pineau *et al.* 2003], exist and have achieved some success, they are distinct from classical planning methods. These approaches work directly within the original MDP framework, which means they still face challenges related to sparse rewards and the need for ongoing interaction with the environment to gather data. However, we seek to leverage classical planning approaches, where an accurate symbolic representation of the environment allows for solving long-horizon tasks without continual interaction. With the right classical planning representation, we can bypass the limitations of sparse rewards and real-time data gathering, allowing efficient solutions to complex problems.

While classical planning methods can address long-horizon tasks, they face challenges in partially observable environments. Similarly, RL policies, while capable of exploration and handling uncertainty, can struggle when domains become too large, as the exploration space may become computationally prohibitive. This suggests that combining planning and exploration strategies could offer advantages for tackling complex long-horizon tasks by leveraging both the structured approach of planning and the adaptive exploration of RL. Classical planning in our problem setup is limited because

the agent cannot interact with the environment to gather additional information. On the other hand, reinforcement learning (RL) allows the agent to interact with the environment, learn, and adapt based on real-time feedback. Given these strengths, we explore combining classical planning and RL to leverage the benefits of both approaches. This combination allows us to overcome the limitations of each method individually.

Our framework assumes the agent is equipped with a set of options, which are also encoded in PDDL. This is because options are effective for handling long-horizon tasks. These options can either be hand-coded or learned. Representing these options symbolically presents a challenge, though it is manageable when the number of symbols is relatively small compared to the complexity of the environment. In classical planning, the number of operators is usually not large, but the number of objects can be significantly greater, further complicating the problem.

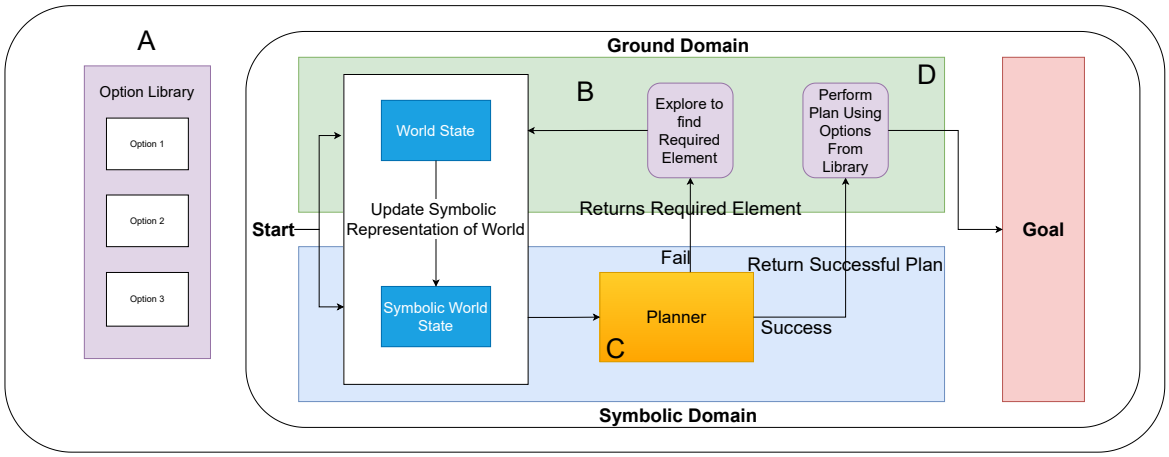


Figure 4.1: *Plan, then Explore* Framework to complete a given goal. The initialisation wherein the symbol state reflects the information extracted from the world state. The planner attempts to find a plan that is successful, if not, an element required to complete the goal is returned where we explore and find the required element using the options selected from the options library.

As seen in figure 4.1 we can view specific sections labeled (A) to (D) being the main sections that contribute to the inner working of the Plan-Explore framework.

4.1 Options

Konidaris and Barto [2007] makes use of predefined options in completing tasks, where as Konidaris and Barto [2009] learn these options through RL techniques. If we do not have options suitable for completing sub-goals we can follow inspiration from both Stolle and Precup [2002] and Konidaris and Barto [2009] and learn the required options. We also need to assume that there is some labeling function that can map the current observations taken from the agent within the ground domain to the correct symbols defined in the symbolic domain. A similar approach to this is seen in Icarte *et al.* [2022] where a labeling function is used to map the state space of the environment

to a set of labels that correspond to different components in the reward function. Due to the restrictions of planning, we require a clear goal specification as well as clearly defined operators with deterministic effects on the environment. Section (A) describes the option library an agent would have access to. This library should be treated as a library of "Black box" options where we do not require any knowledge of how the sub-goals of these options are achieved. We only care about the final outcome of these options and the final effects they have on the environment. Thus knowing the initial state and the outcome we can know the exact outcome the option would have on the environment allowing for efficient planning.

4.2 Ground and Symbol State

Section (B) describes the observations derived from the ground domain, which represent the MDP states. For instance, in Minecraft, this would correspond to the pixels of the agent's point of view (POV). These observations are symbolically represented within the symbol domain, where the symbols are updated at each time step based on the agent's current observations. We use a labeling function to convert specified observations taken by the agent about the current environment and translate these observations into symbols that are usable by the planner. For example, if we are using our labeling function with an observation of the agents (x, y, z) coordinates, the labeling function would translate the given state into the symbolic form being:

```
(at-x ?ag nx)
(at-y ?ag ny)
(at-z ?ag nz)
```

which can be read as agent ag is at the x coordinate position nx, y coordinate positions ny and z coordinate position nz.

This symbolic form shows that given observation Z we can denote the feature vector for the observation Z as $f(Z)$. Next we map this feature vector $f(Z)$ using the mapping function M to symbols in the symbol domain P such that $M : f(Z) \rightarrow P$. Now that we are capable of obtaining the symbols from observations using the mapping function M we need to update the symbol domain given the observation Z. Let P_t be the symbols in the symbol domain at time step t, and let P_{t+1} represent the newly updated symbols after mapping observation Z at time t. We can express this function as follows :

$$P_t \cup M(f(Z)) \tag{4.1}$$

P_t is the set of symbols extracted by $f(Z)$ in the pixel domain and mapped using M for the observations Z taken from the environment. If an extracted symbol from the pixel domain has not been seen or represented in the symbolic domain, the new observation is utilized to create a new symbol. This process allows exploration in the ground domain to be accurately reflected in the symbolic domain.

4.3 Planning

(C) signifies the planning stage of the framework. Any planner that supports symbolic representations will suffice. For example, Fast Downward [Helmert 2006] is compatible with PDDL2.2 [McDermott *et al.* 1998]. Therefore, to use Fast Downward as the planner of choice, the PDDL2.2 format must be utilized.. These representations can be used in conjunction with the symbolic world state to plan for a given goal. This planner has two possible outcomes, a successful plan that can be followed to complete a given goal or a Fail. In the case of a failed plan, instead of discarding it entirely, our approach enables the planner to retrieve a useful element from a failed plan, both from the world state and the symbolic world state, in order to move closer to the desired goal. For example, if we are trying to construct a stone column within a 3D environment, if the agent does not hold any stone the planner would return a required element of stone. This returned element sets a new sub-goal that the agent needs to achieve before continuing with the main goal for completing the complex structure. Fast Downward has had extensive development and proven effectiveness in various planning domains with a strong track record in the planning community as well as the ability to use various heuristics in order guide the search process. Due to these facts we use Fast Downward as our planner of choice.

4.4 Exploration

(D) represents the ground domain where the agent interacts with the environment, exploring the partially observable world by enacting actions. If the planner successfully finds a viable plan, the corresponding options are executed to progress towards the goal. These options are guided by the symbolic world state, allowing the agent to follow the predefined course of actions that lead closer to the goal.

4.4.1 Plan-Explore

Algorithm 2 depicts the high-level overview of Plan-Explore. Given a goal specification G , mapping function M , update function U, options O we begin the algorithm with an initial mapping from the ground state to the symbol state is performed using the mapping function specified in Equation 4.1.
While we have not achieved the goal represented symbolically as G, we enter a continuous cycle where we update the symbol state with the newly found observations using U. The planner then attempts to plan using the current state of the ground domain represented symbolically and the goal specification. If the planner fails to find a viable plan to achieve goal G, we extract the required object from the planner.

When we reference extracting the required object we can view the *CustomPlan* in Algorithm 2 on line 25. Within this procedure, we make a copy of the goal specification, locally, allowing editing of the goal specification without affecting the overall goal specification. We then attempt to plan with the current symbol space. If the planner fails to find a successful plan, we remove a predicate from the goal specification, which is represented as a conjunction of predicates, and save this predicate. We then update

Algorithm 2 Plan-Explore

Mapping function M
Feature vector f
Update function U
Options $O = \{PlaceBlock, PickupBlock, \dots\}$
Goal specification G
World Observation o
Current State P

procedure BUILD STRUCTURE
    o = make world observation
    Initialise: P using $U(o)$
    while True **do**
        Call CUSTOMPLAN(P,G) ▷ Call the Plan function
        if Plan returns "Success" **then**
            Perform successful plan using O
            END
        if Plan returns "Fail" **then**
            Required-Object = Decode Planner Output
            o = Call EXPLORE(Required-Object) ▷ Call the Explore function
        Update P Using $U(o)$

procedure EXPLORE
    Input: Required-Object
    Output: world observation
    Explore to find Required-Object returned by planner
    return new world observation

procedure CUSTOMPLAN
    Input: Symbol Space P,
    Input: localized Goal specification G
    Output: (PlanState,b)
    Current required object b
    PlanState = Plan using P
    while PlanState = "Fail" **do**
        b = remove object from G
        update goal of P with G ▷ G has one less object
        Plan using P
    if b is not empty **then**
        return ("Fail",b)
    return ("Success",b)

the symbolic goal state and re-plan. If multiple re-plans are required, we continue removing predicates from the goal state iteratively until a plan is found that satisfies the goal. Once the planner returns a successful plan, we assess whether we have a required

predicate stored. If so, we decode it, since it is in PDDL format, which the agent cannot directly use. This process is illustrated in Figure 4.2, where we iterate through the planning process, removing predicates one by one until we achieve a feasible plan. When the deletion of a predicate results in a successful plan, we use the recently deleted predicate as a target for exploration. The agent then explores the environment until it locates the object referenced by the predicate.

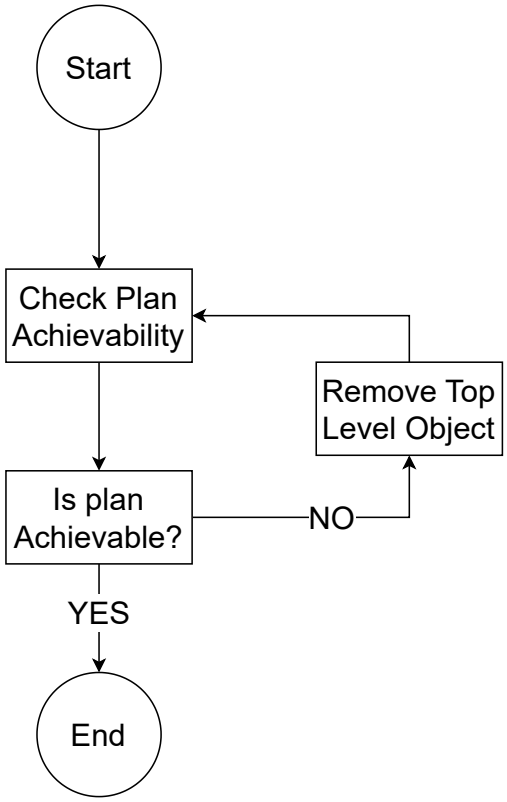


Figure 4.2: High Level overview of the symbol extraction process when planning

When doing this we have a choice of removing a singular object from the goal specification G or multiple objects. This means that for the case of removing a singular object, we continually plan while removing an object for each plan iteration. We then save the most current object in a placeholder and remove this object from the local goal specification. At the eventual success of the planner, which we still consider an overall fail in terms of the global goal specification, and knowing which object is required, we are able to explore for the required object in the ground domain. The exploration process can utilize methods such as random option execution, where options are selected randomly, continuing until the required object is found. We continue this process of updating the symbol space and planning until we get a success without a returned object. This shows that the planner is able to return a successful plan that completes the goal specification G.

The implementation of the planning sequence wherein we continually remove objects from the goal specification between each planning instance in order to find the current, most important object to complete the goal specification is one of our contributions to the field of planning and planning in partially observable environments.

Having shown the framework designed to handle complex tasks in both full and partially observable object-oriented domains we next test this framework in a 3D, open world environment Minecraft, which is a partially observable state space.

Chapter 5

Applying Plan-Explore to Construction Tasks in Minecraft

Minecraft is a popular sandbox video game known for its open-world environment, with unique characteristics such as various materials with unique interactions and a procedurally-generated world. Minecraft can be seen as a partially observable environment due to an agent's view and knowledge of the world being limited, by aspects such as terrain that obscures vision as well as limited information about the surroundings coupled with uncertainty about the state of distant occluded regions. The combination of partial observability, infinite state space and dynamic elements creates a challenging test bed for various algorithms and methods. Due to Minecraft being a complex 3D, partially observable environment, it serves as an ideal environment to test our framework.

Making decisions within the partially observable environment (ground domain) of the agent comes with various complications. Since the environment is partially observable, the agent does not have complete information about the state and has to make decisions using this partial information. These partial observations often contain noise and uncertainty. Given the high volume of data represented within a single state in the ground domain, these states can include unnecessary information, further complicating the decision-making process and hindering the ability of an RL agent to learn effectively.

Uncertainty from environmental changes or stochasticity of the environment such as slight differences within the point of view of the agent can further complicate the learning process.
In a partially observable environment, effective decision-making requires the ability to remember past observations over time. Since Q-learning or basic deep Q-networks (DQNs) may struggle to capture long-term temporal dependencies, this could hinder their performance in environments with partial observability. Since we can highlight the struggles planning and RL face in the domain of Minecraft we can conclude that Minecraft is a suitable test environment to test the Plan-Explore framework outlined in Chapter 4.

Some obstacles experienced by RL algorithms in a partially observable environment can be avoided if decisions are made in a concise and discrete space such as the symbolic domain. Using the symbol domain as a platform for reasoning and exploration of the environment will result in a more concise choice of options for the agent as the symbolic representation abstracts away the low-level irrelevant details. Since a full symbolic domain is almost impossible to achieve in such a complex environment, the agent will only be able to use a partially observable symbol space for reasoning and decision making, but since the symbolic domain can capture the essential aspects of the environment in a concise manner, this can help filter out irrelevant details and reduce noise, providing a more focused and less noisy representation of the environment compared to ground data.

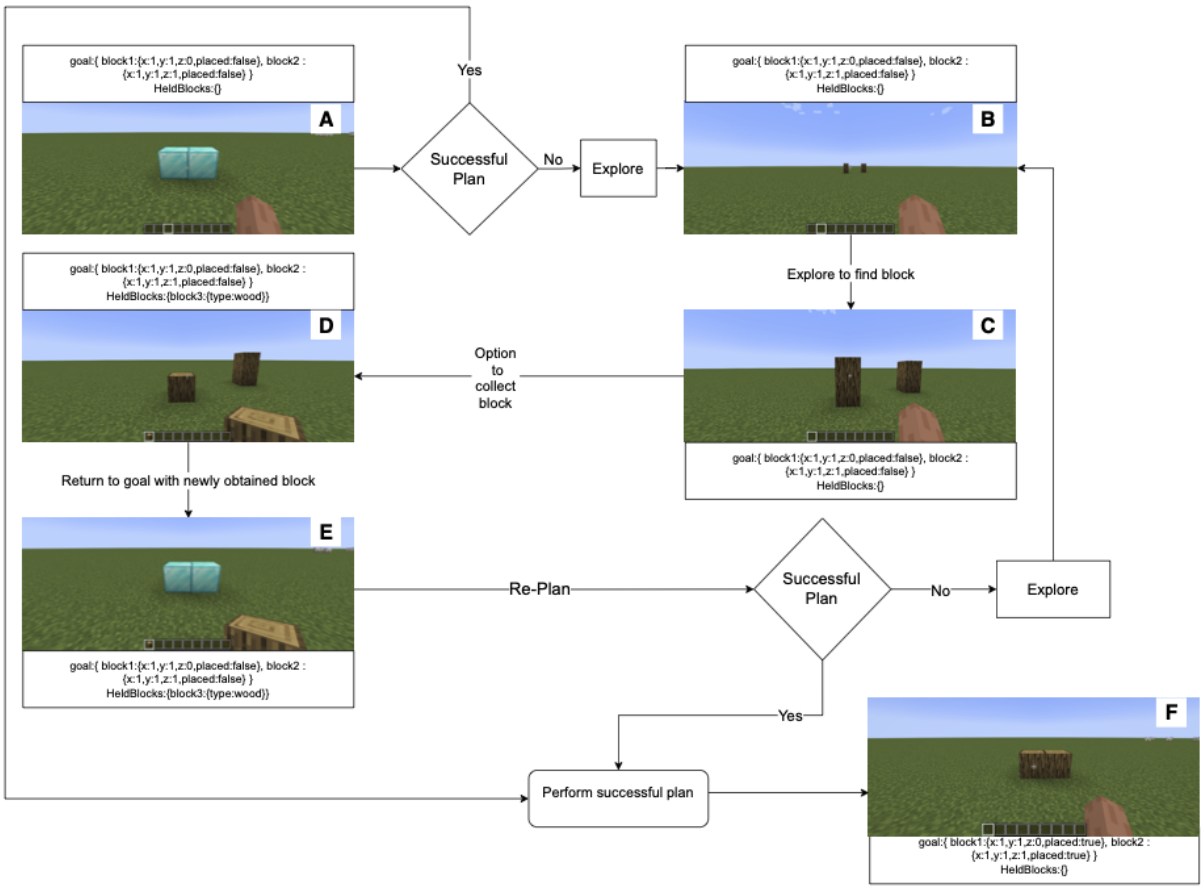


Figure 5.1: The procedure followed to create a structure of two blocks. The goal is to collect and place two wood blocks in the locations of the diamond signifier blocks.

Viewing Figure 5.1, the goal is to collect and place two blocks in a designated location with available options of moving to a location, collecting a block and placing a block. Initially, when an agent is placed into the partially observable world, the agent can only gain knowledge about it's immediate surroundings, and thus the symbolic domain will only hold data that is observed by the agent.

The planner then plans to complete the task with the known information in symbolic form. If the plan is unsuccessful the required block is returned, after which, the agent

explores using the logic defined in Algorithm 2 on line 20 to find the required block which is reflected in Figure 5.1 (B) and (C).

When the agent executes an action, the predicates of the symbolic domain are updated. Therefore, as the agent acts within the environment the symbol space is updated with relevant information about the agent's surroundings. We can view this update in figure 5.1 (D) where the agent collects a wood block, the *HeldBlocks* symbol reflects the new block the agent holds after actions and options were performed within exploration to collect this block.

Once a block has been collected we re-plan. If the plan is successful we can use this plan to complete the goal reflected in figure 5.1 (F), if the plan is unsuccessful we repeat flow until this plan is successful.

We compare our framework to two baseline algorithms, namely DQN and a DQN with options. The basic DQN uses only primitive actions to achieve tasks within Minecraft, where as the DQN with options only uses options to achieve set goals. Using a model trained on using high-level options such as searching for a block or collecting a block which requires searching and picking up a block, will serve as a much closer comparison for the use of planning. We use the basic DQN algorithm and DQN with options to serve as RL and hierarchical RL baselines respectively.

5.1 State Space

5.1.1 Pixel State Space

Using the MineRL Framework, the state space S contains three different observations from the Minecraft environment, these being the location coordinates, point of view (POV), and the inventory of the agent.

The location coordinates represent the position of the agent within the Minecraft world. Typically, location coordinates consist of three values, these being the x-coordinate, y-coordinate and z-coordinate. The x-coordinate represents the horizontal movement through the environment, which would be the east-west movement. The y-coordinate represents the vertical position of the agent and the z-coordinate represents the position in the north-south direction.

The point of view (POV) of the agent encapsulates its perception and interaction within the partially observable Minecraft world. This observation is how the agent perceives the environment and enables the agent to interact with its surroundings. This POV is returned as an RGB image of the agent's first person perspective, where the image returned is the size of 500x500 pixel square.

The agent's inventory is a dictionary that holds any possible item that exists in Minecraft. This means that for each item the agent is currently holding, there is a key-value pair with the name of the item and the amount of that item the agent is currently carrying. When picking up and dropping items, the counts for the valid keys increase and decrease respectively with the number of items carried never decreasing below zero.

Due to these observations being different spaces, we use a convolutional neural network for the point of view of the agent, and combine a vectorization of the inventory and the location vector into one large vector that can be used as input for a neural network. This can be seen in Figure 5.2, which displays the makings of the convolutional network and that the vectors are combined before they are used in the fully connected layers.

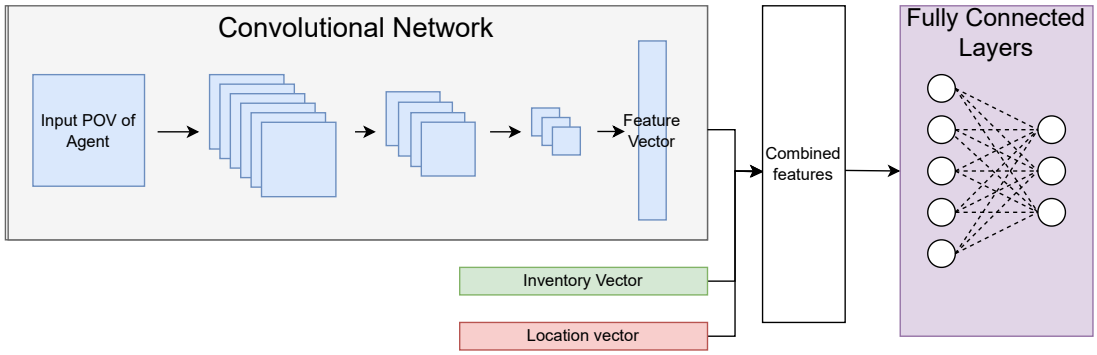


Figure 5.2: Feature combination used in the DQN Algorithm

5.1.2 Symbolic state space

Using the Planning Domain Definition Language (PDDL) as our framework, we adopt a symbolic representation approach to define our state space within our planning domain. We use a similar symbolic representation to Hill *et al.* [2023].

Since our planning domain is represented in PDDL format, we define various types of object and entities that are relevant to our planning domain where each type follows a format of specifying the type name followed by a list of sub-types.

```
(:types
locatable int - object
agent item block - locatable
destructible-block - block
wooden-block wooden-planks  - destructible-block
destructible-item - item
position count - int
)
```

Referring to our type definitions above, we can view that there is a declaration of two objects, locatable and int. These objects serve as identifiers for noting sub-objects as either locatable objects or denoting an object as an integer value. When doing propositional planning we are unable to make use of fluents which are a mapping of objects to a numeric value. This meaning that the only way to use integers is to represent integers as objects with predicates such as the predicate *sequential* which defines the sequential nature of two objects.

We define the agent, item and block types with the locatable type serving as a subtype for agents and blocks which indicate the spacial presence within the environment.
We also instantiate the destructible-block type as the type of block, signifying that a block in our environment is destructible. Using the destructible-block as a type to define the different types of blocks that are possible in our environment being wooden-block and wooden-planks, this then maps that wooden-block and wooden-planks are destructible-blocks of type block which is a locatable object. Lastly, we define types of position and count with the type of int signifying that we have two objects that are directly correlated to integer values.

```
(:predicates

    (block-present ?b - block)
    (agent-alive ?ag - agent)
    (neq ?x1 - int ?x2 - int)
    (are-seq ?x1 - int ?x2 - int)
    (agent-has-n-wooden-blocks ?ag - agent ?n - count)

    (at-x ?l - locatable ?x - position)
    (at-y ?l - locatable ?y - position)
    (at-z ?l - locatable ?z - position)

    (on-top-of ?b1 - block ?b2 - block)
    (on-floor ?b - block)
    (agent-on-floor ?ag - agent)
)
```

In the symbol space we decompose the observations outlined above and represent them symbolically as objects and predicates. We can directly correlate the coordinate location observation as a predicate for location in the symbol space. This means that any loadable object like the agent, or blocks, can have its current location being (x, y, z) written as three predicates namely :

```
(at-x ?l - locatable ?x - position)
(at-y ?l - locatable ?y - position)
(at-z ?l - locatable ?z - position)
```

When reviewing a single predicate from above such as *(at-x ?l - locatable ?x - position)*, this represents an object's x coordinate in the pixel state. This predicate can only be defined for any object that has a type of locatable, so for our implementation of the symbol space, only the agent, items and blocks are locatable. Given these restrictions, any block or item found has its location defined in the symbolic domain.

There are a few predicates to define the environment in a symbolic format. These logical statements or conditions describe the state of the Minecraft environment in a symbolic format. There are a few predicates that define simple objects' states within the environment, these predicates are :

```
(block-present ?b - block)
(agent-alive ?ag - agent)
(on-top-of ?b1 - block ?b2 - block)
(on-floor ?b - block)
(agent-on-floor ?ag - agent)
```

These predicates define simple environment states such as if the agent is alive in the environment, if the agent is on the floor or ontop of another block, if a blocks is present within the environment and if a block is on-top of another block instead of being on the floor. For example if we view Figure 5.4 we have two blocks on the floor and two blocks on-top of other blocks. Focusing in on block1 (Bottom left), we can view the symbolic representation of the block which reflects the predicates relevant to this block.
Other predicates such as the *(agent-on-floor ?ag - agent)* and *(agent-alive ?ag - agent)* are specifically designed to capture the agent's game state. Lastly predicates such as :

```
neq ?x1 - int ?x2 - int
(are-seq ?x1 - int ?x2 - int)
(agent-has-n-wooden-blocks ?ag - agent ?n - count)
```

are logical statements to define integer objects. These statements capture information such as if two objects are not equal, or two objects are sequential. Lastly, the predicate *(agent-has-n-wooden-blocks ?ag - agent ?n - count)* defines the number of a block type the agent currently contains.

Since our planner of choice cannot handle integer operators, we circumvent this issue by defining integer objects and manually defining their sequential nature allowing us to effectively use objects as a form of integer system. For example, if we define objects of type int namely *n1, n2, n3, n4* we can define a sequential nature by defining :

```
(are-seq n1 n2)
(are-seq n2 n3)
(are-seq n3 n4)
```

5.2 Action Space

The action space encapsulates any possible action the agent can take at any given time. Within the domain of Minecraft, these actions can vary from the simple agent movement of forwards, backwards, left and right to more complex actions such as the camera control and tool selection within the inventory of the agent. In our domain of Minecraft we limit the types of actions an agent can perform and we can view these actions in Table 5.1 below.

When learning or designing options we use the primitive actions defined in Table 5.1 as the building blocks for options that achieve sub-goals of *Move to a location*, *Pick up block* and *Place block*. These options are defined in Table 5.2 with a descriptor on what primitive actions are required to achieve the sub-goal defined.

Table 5.1: Agent Actions in Minecraft

Action	Description
Move	Cardinal movement (North, South, East, West)
Look Up	Increase the vertical rotation of the agents viewpoint by 30°
Look Down	Decrease the vertical rotation of the agents viewpoint by 30°
Place	Place the currently held block in facing direction
Destroy	Attack in the direction the agent is facing
Pickup	Cardinal movement (North, South, East, West)

Table 5.2: Agent Options in Minecraft

Option	Description
Move to location	Cardinal movement (North, South, East, West) to a given location
Pick up block	Destroy action and Visual orientation (up and down) [1]
Place block	Place action and Visual orientation (up and down)

5.3 Options



Figure 5.3: The option displayed shows placing a block at a specific location. This requires movement, item selection and the action of placing a block.

Options within the pixel domain can be viewed as sub-policies learnt from focused training. Our options are RL policies that are learnt using the basic DQN. Hyper parameter tuning was performed while learning policies to learn an optimal policy for each option. These hyperparameters and their values are defined in Table 5.3 .

When learning to move to a location we can adapt the reward function as well as the actions available to the agent to achieve the desired goal. This allows us to learn efficient options that can be used by both the DQN with Options as well as the Plan-Explore framework. We follow this format to learn the options defined in Table 5.2, when learning the *move to location* option we limit the observation to only agent co-ordinates. These pre-learned options, stored as policies, are directly accessible to the DQN with Options.

[1] Visual orientation represents the angle at which the agents visual field is tilted, where directly ahead is 0°, directly down is 90°and directly up is -90°

Table 5.3: Hyperparameters and their Values

Hyperparameter	**Value**	**Tested**
Replay Buffer size	1×10^4	[1×10^2, 1×10^3, 1×10^4]
Learning Rate	1×10^{-4}	[1×10^{-3},1×10^{-4},1×10^{-5}]
Discount Factor	0.90	[0.80, 0.85, 0.90, 0.95, 0.99]
Batch Size	128	[64, 128, 256]
Learning frequency	1	[1,2,5]
Target Network Update Frequency	150	[100, 150, 200]
ϵ decay percentage	0.9	[0.7, 0.8 , 0.9]

When an option is enacted, its corresponding sub-policy is activated, which executes a sequence of actions to achieve a specific goal. During this process, the main decision making process pauses, waiting for the completion of the sub-policy. Once the sub-policy successfully completes, control returns to the main process, which can then continue with the next step in the decision-making process. This allows for modular and efficient task execution within the overall framework of the DQN.

5.3.1 Rewards

Rewards for both DQN and DQN with options vary according to the task. For all tasks learnt by the basic DQN, as well as the DQN with options, each task has a reward of *-1* for every time step the goal has not been reached. This reward is to influence learning an optimal policy that completes the given task in the shortest amount of time steps.

When learning the option of moving to a given location a negative reward equal to the distance the agent is from the final goal with a very large positive reward when the goal has been reached. This negative reward is represented by:

$$R(x, y, z) = -k\sqrt{(x_g - x_a)^2 + (y_g - y_a)^2 + (z_g - z_a)^2} \tag{5.1}$$

This shows that with locations (x_a, y_a, z_a) for the agents current location and (x_g, y_g, z_g) for the goal location, the larger the distance between the agent and the goal the larger the negative reward is. This coupled with a constant negative reward of -1 for each timestep influences the learning of a policy that achieves reaching the goal location in the shortest amount of time steps.

As above when learning the option for placing a block, a positive reward equal to the total step limit for the episode (being 10) is given and a negative reward of -1 for each step taken within the environment. This influences the agent to learn a policy where the agent places the block as fast as possible. A similar reward structure is used for learning the option of collecting a block, the increase in inventory items for a specific block results in the final goal being reached, where the final reward is equal to the total step limit of the episode. This completion reward being a value of 10 is given with the addition of a negative reward of -1 for each step taken within the environment.

Since Plan-Explore does not use any learning in executing the final goal and all options are written no form of reward is implemented.

5.4 PDDL Operators

Making the assumption that options in a symbolic format can be written down, we can view the effect of the option as updating the agents location from its current location to the goal location.

Viewing the PDDL operator,

```
 (:action move-to-location
:parameters (?ag - agent ?x_start - position ?x_end -
   position ?y_start - position ?y_end - position ?
   z_start - position ?z_end - position)
:precondition (and
    (agent-alive ?ag)
    (agent-on-floor ?ag)
    (at-x ?ag ?x_start)
    (at-y ?ag ?y_start)
    (at-z ?ag ?z_start)
    (not (exists (?bl - block) (and
        (block-present ?bl)
        (at-x ?bl ?x_end)
        (at-y ?bl ?y_end)
        (at-z ?bl ?z_end)
    )))
)
:effect (and
    (not(at-x ?ag ?x_start)
    (at-y ?ag ?y_start)
    (at-z ?ag ?z_start)
    )
    (at-x ?ag ?x_end)
    (at-y ?ag ?y_end)
    (at-z ?ag ?z_end)
)
)
```

we can make the broad observations such that the agent has to be alive and on the floor, the agent must be in the starting position which is the current location of the agent, and a block cannot exist at the final location the agent needs to move to.

The *Pick up block* option used in RL is learnt through the same technique as the *Move to location* option where a basic DQN learns to pickup a block directly in front of the agent. the actions available to this agent are the *Look Up*, *Look Down*, *Destroy* and

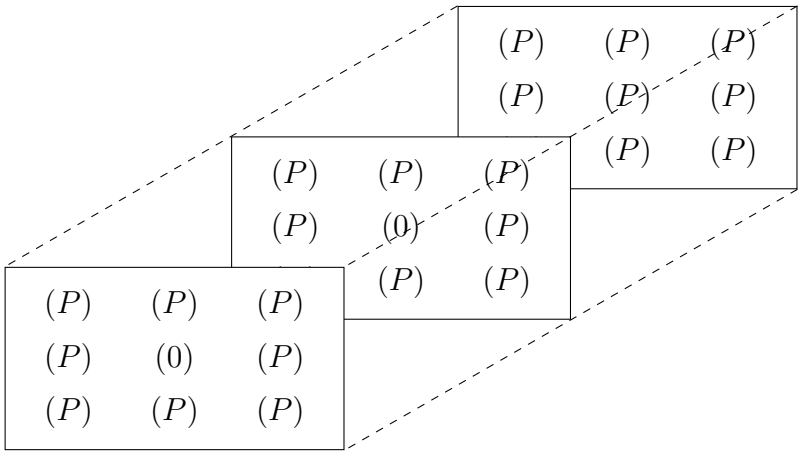
Pickup actions. This guarantees that the agent learns a policy in which it successfully picks up a block. Appendix B.1.13 shows the PDDL operator in which an agent will pick up a block with the final outcome being the same as the option learnt by the DQN.

The *Place block* option holds similar actions the *Pick up block* option has with differences in a substitution of action from *Destroy* with *Place* and the removal of *Pickup*. The DQN receives the same observations as before with the addition that the agent is already holding the required block. Appendix B.1.14 reflects this option in the form of a PDDL operator similar to Appendix B.1.13 but instead of increasing the number of blocks held by the agent and asserting that the block is not present in the environment, the number of blocks is decreased and the block location is updated and asserted to being present in the environment.

Since we only have a total of three options and writing these options down symbolically is not too much of a burden. These options are used by both DQN models as well as the Plan-Explore framework to accomplish tasks in the Minecraft environment.

5.5 Representation of Goal

The goal is represented as a 3D vector containing information that distinguishes the type of block. In the context of the RL model, this goal representation is used as part of the input to the neural network (NN). Specifically, the NN receives a flattened version of this goal. The vector serves as a feature in the NN's layers, allowing the model to learn the necessary actions to achieve the goal. The representation of this vector is as follows, $\langle X, Y, Z \rangle$ where $\langle X, Y, Z \rangle$ represents the coordinate set for a singular block. We denote different block types such as P for planks and any empty space as 0. For example, we can represent a structure with a floor, four walls and a singular blocks space withing the center of this structure as follows :



With the corresponding coordinates $\langle X, Y, Z \rangle$ are as follows.

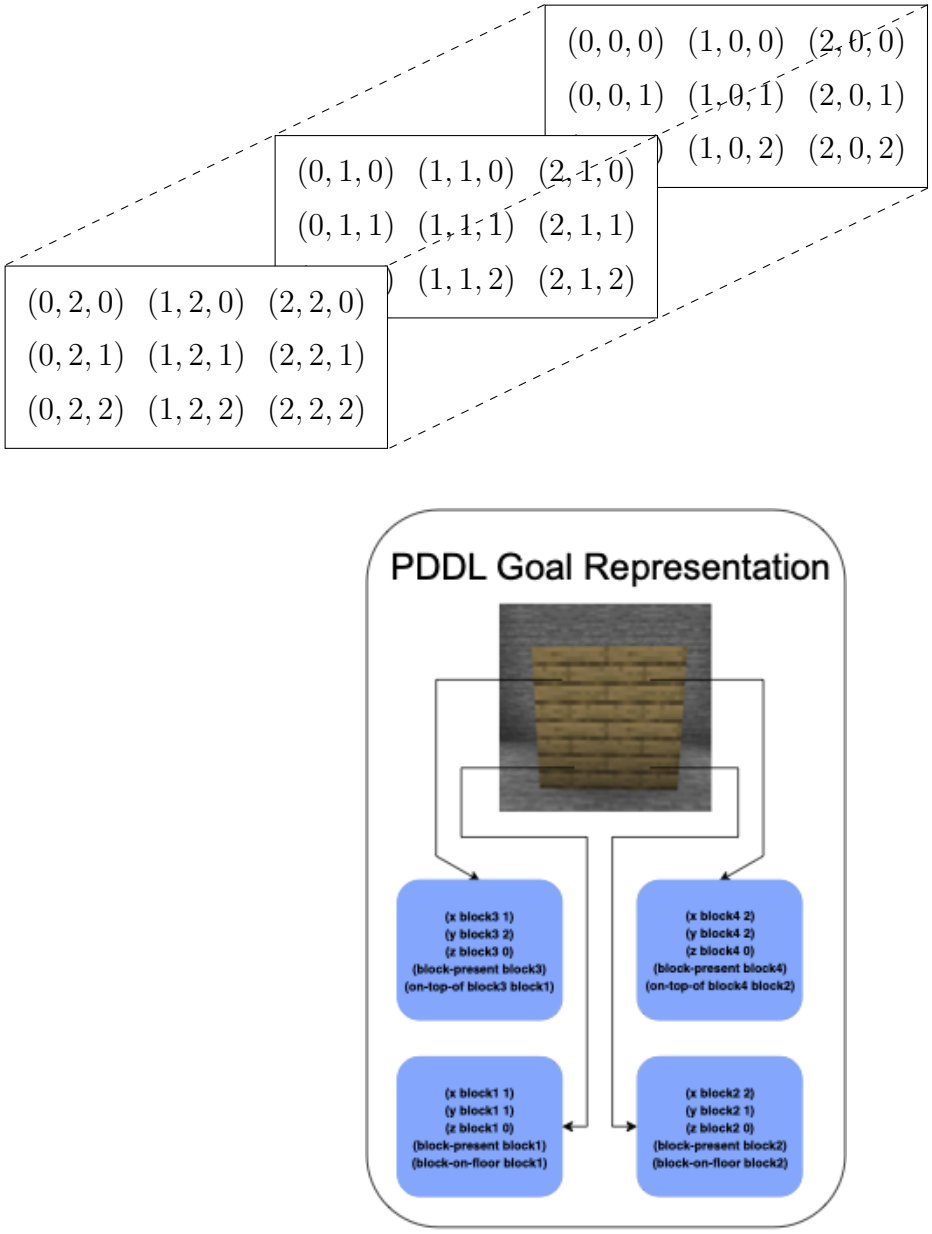


Figure 5.4: A direct comparison between the PDDL representation and the world state, showing how blocks in Minecraft are represented in a symbolic manner. This image represent a structure of four blocks where each block is translated into a symbolic object. Each block represented symbolically is comprised of the x, y, z coordinates of the block location relative to the Minecraft world coordinate system, as well as predicates expressing whether the block is present within the world and if the block is on the floor or on top of another block.

The representation of the goal within the symbol domain varies to that of the pixel domain. The initialisation of the goal is identical to that of the pixel domain as seen above with the point cloud representation, however this is then converted into a text based representation as seen in Figure 5.4. Due to the nature of the designed symbolic domain we distinguish between a block on the floor and a block on a block. We also

specify whether a block is present in the world as a defined block can be defined but held by the agent thus not currently existing in the world.

5.6 Environment

The environment that is used for all training and testing is the MineRL Framework [Guss *et al.* 2019]. Competitions stem from this framework such as Guss *et al.* [2020] which promotes the use of human demonstrations to reduce the number of samples needed to solve complex, hierarchical, and sparse environments. This environment allows for an agent to make similar observations and actions as that of a user. This environment also supplies specific sub environments that allow for option learning such as navigation, mining and crafting.

5.7 Experiment Setup

The setup for the RL models involved a deep Q-network (DQN), with a distinction between the standard DQN and the DQN with options. The key difference lies in the actions and options available to each model, which are used as benchmark algorithms to compare the performance of the developed Plan-Explore framework. The Plan-Explore framework was provided with both primitive actions and options.

We performed tests for a multitude of tasks starting from the simplest task and increasing the complexity of the tasks to display the capabilities of each model. The tasks the agent was required to complete include:

- Place a singular block. This requires the finding, collecting and placing of a singular block in a designated location as seen in Figure 5.5.
- Place 2 blocks in a row, requiring the agent to explore and collect two blocks and place them in the desired location, depicted in Figure 5.6.
- Place 4 blocks as a wall. The increase in complexity comes with the increase in number of blocks required for the structure and an increased number of goal locations in the (x, y) axes. This task is depicted in Figure 5.7.
- Place 4 blocks for a floor. This task has an increased number of goal locations along the (x, z) axes requiring the agent to move more within the environment. This task is depicted by Figure 5.8.
- Place 6 blocks as a wall. We continue testing with an increasing number of blocks within a wall format displayed in Figure 5.9.
- Place 8 blocks as a wall. Increasing the difficulty with added blocks. This task is displayed in Figure 5.10.

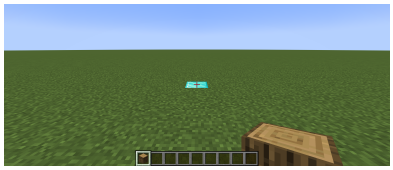


Figure 5.5: Singular Block



Figure 5.6: Two Blocks



Figure 5.7: Four Block Wall



Figure 5.8: Four Block Floor



Figure 5.9: Six Block Wall



Figure 5.10: Eight Block Wall

Figure 5.11: The various goal states used for testing

5.8 Metrics

When testing the various methods for comparison we cannot compare wall clock times as the comparability between planning and RL does not directly translate. The variability in each method's parameters makes clock time an unreliable measure when comparing the performance of our framework to learning-based methods. However, for evaluating planning alone, wall clock time is sufficient to draw meaningful conclusions. Instead for comparison with learning we opted for the total number of environment interactions the agent makes. This being that every primitive action taken within the environment is counted towards the total number of environment interactions the agent enacts to reach the final goal. For DQN and DQN with options this includes all episodes taken to train a policy sufficient in completing the given goal, an alteration occurs to the calculation when dealing with options. For each option that is used, for example, moving to a location or picking up a block, the corresponding total number of interactions from the training of these policies is used in the calculation. We set a maximum number of environment interactions as a cutoff to conclude that it is unsolvable by an algorithm. This cutoff was set at two million environment interactions.

5.9 Summary

In this chapter, we evaluate the performance of our framework, defined in Chapter 4, by comparing it against RL methods, specifically a DQN and a DQN with options, using Minecraft as the test environment. Minecraft's partially observable and complex nature makes it an ideal setting for these experiments. We highlight how options are

integrated: in the pixel domain, options are executed based on a policy learned using RL for the DQN, while in the planning domain, they are explicitly defined. These options are assumed to be provided to both the DQN and our framework.

We begin by explaining the implementation of our framework within the Minecraft environment, using Figure 4.1 to illustrate its workings. Section 5.1 discusses the state space, followed by an exploration of how PDDL is used as the symbolic representation for our planning domain. We emphasize the key features of the symbolic state space that PDDL offers.

In Section 5.2 , we describe the primitive actions available to a basic agent, as well as the options used by both the DQN with options and our framework. These options, as detailed in Section 5.3, are explicitly defined for both RL and planning tasks. We then turn to the PDDL operators in Section 5.4, which represent the handwritten options within the symbolic domain.

Next, we define the goal representation in both the pixel and symbolic domains, with specific attention to how the block states are represented in the symbolic space. We also describe the MineRL framework used for the experiments, discussing the advantages it provides for this type of research.

Finally, in Section 5.7, we outline the experimental setup, including the gradual increase in complexity by expanding the number of blocks within the goal state.

Chapter 6

Results

The following section presents the comparative analysis between the three algorithms discussed, which include a basic DQN algorithm that only has access to primitive actions, a more advanced DQN that has access to options defined in Section 5.3 and the Plan-Explore framework designed to use the benefits of both planning and options to achieve tasks in a partially observable environment formally described in Chapter 4.

For a fair comparison, it is crucial to choose metrics aligned with each approach's objectives. In reinforcement learning (RL), the emphasis is on sample complexity—the number of interactions or samples the agent requires to learn an effective policy. This metric captures the efficiency of learning in environments where data is costly or time-consuming to obtain.

In contrast, classical planning typically focuses on wall clock time, as it often involves deterministic search strategies where computation time directly reflects the efficiency of finding an optimal solution. The distinct goals and computational methods of RL and planning make direct comparisons based solely on wall clock time unreliable and potentially misleading.

Within our framework, we determine the number of blocks to collect between each planning iteration, as shown in Algorithm 2, line 20. The number of blocks varies depending on the goal to reduce the planning cost at each phase. Since we evaluate performance based on the number of environment interactions rather than wall clock time, adjusting the number of blocks helps manage the planning overhead. Therefore, when comparing these algorithms, we consistently use the number of environment interactions as the key metric.

When counting the number of environment interactions for the basic DQN, we keep track of every action the agent takes throughout training until an optimal policy is found. This is also the case when we learn options through the basic DQN as we require those values for analysis. When tracking the number of environment interactions for the DQN that have access to options, we also track all interactions the agent has on the environment including when exploration occurs. We measure sample complexity by counting each low-level (primitive) action the agent executes in the environment. For instance, if an option involves moving ten blocks north, we count ten interactions, as

each block moved represents a separate primitive action. Our framework tracks these interactions similarly to advanced DQN with options, by accounting for both options and individual actions.

To ensure a fair comparison, we account for the total number of environment interactions required to learn options in the basic DQN. When calculating the interaction count for both the DQN with options and the Plan-Explore framework, we include the interactions used during option learning. This way, we compare the basic DQN's total interactions for completing a task against the combined interactions for the DQN with options, including the sum of interactions required to learn each option. This comparison can be seen in Figure 6.1.

6.1 Sample Efficiency

For reference, the interactions needed to learn the options for specific tasks—*move to a location*, *pick up a block*, and *place a block*—are 65,240, 67,917, and 64,600 interactions, respectively.

As we can see displayed in Figure 6.1, a simple DQN cannot be used to solve a complex task such as building a wall or house. The limitation is reached where a single block needs to be collected and placed. However, the simple DQN is very capable in learning options that can be used in both the Plan-Explore framework as well as the more complex DQN with options. The DQN with options was successful in a few of these tasks but reaches the limit at the task complexity where 8 blocks need to be placed. The Plan-Explore method shows that all these tasks are achievable. Viewing Figure 6.1 and Figure 6.2, we see that although the Plan-Explore framework can handle the complexity of the tasks better than both the DQN and DQN with options, there is a drawback with planning time. The increase in symbols within the domain while planning occurs creates an exponential increase in time taken to plan and account for these new symbols. Viewing Figure 6.1 we see that the Plan-Explore values barely change. This is due to a minuscule increase in environment of interactions compared to that of DQN and DQN with options.

Figure 6.1 demonstrates that the framework proposed in Chapter 4 outperforms both a basic DQN, which fails at completing a single block collection and placement task, and a DQN with options, which can handle tasks involving up to six blocks. An alternative view of this difference is provided in Figure 6.3, which shows that Plan-Explore extends task completion by two additional blocks over the DQN with options.

When testing the limits of planning within this domain displayed by Figure 6.2, we compare planning times for three different complex structures. Viewing Figure 6.2, we see that as the task complexity increases, the planning time for each task increases. For each new block added to the goal state there is a significant increase in planning time. Due to the differences in structure layout we can see that the more complex structure has a significant difference in planning time to the two simpler tasks of stacking blocks and placing blocks in a line. Visuals of the goal structures are provided in Appendix sections A.1, A.2, and A.3. From this, we can conclude that although we eliminate

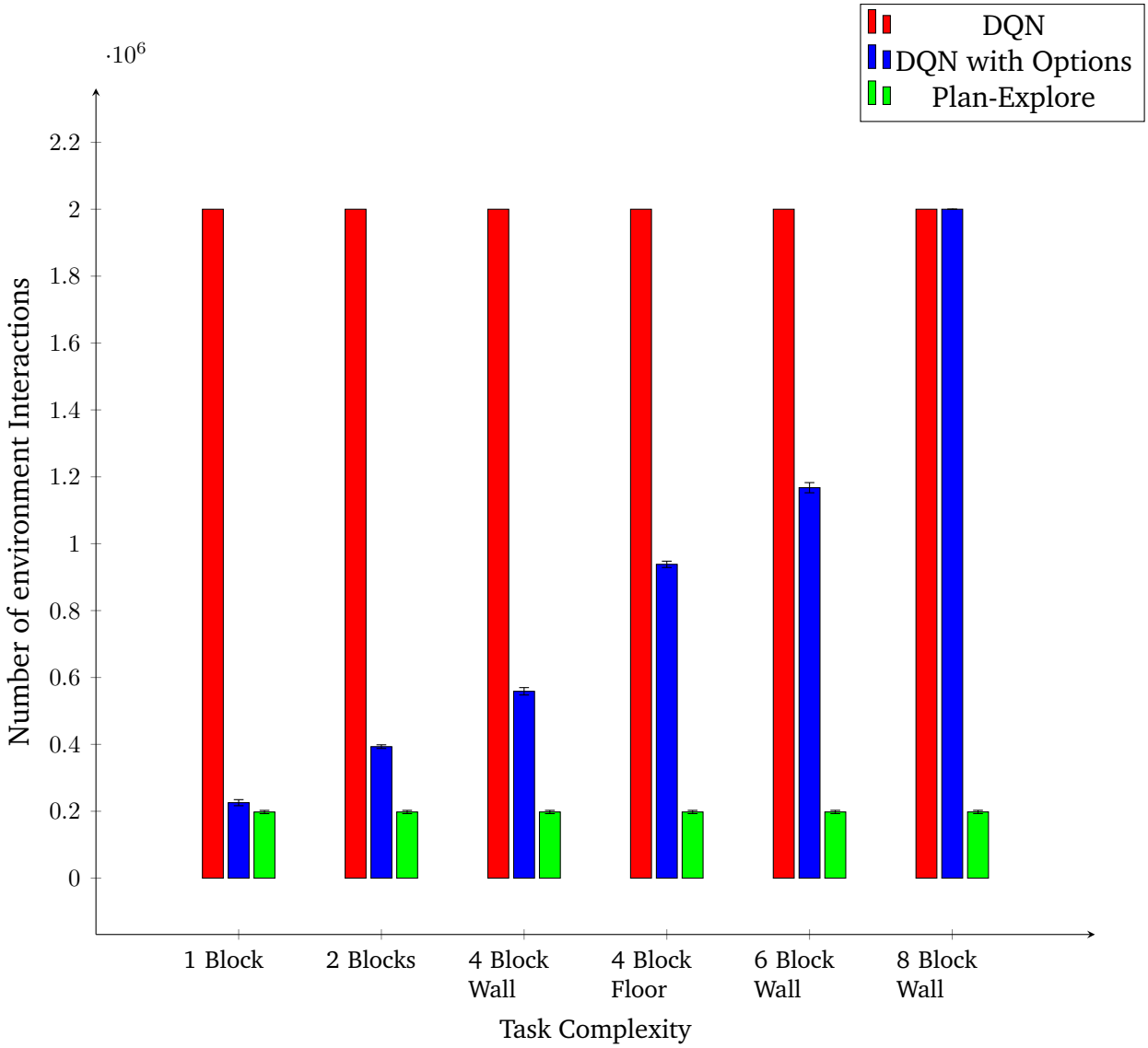


Figure 6.1: A direct comparison between the DQN, DQN with options and Plan-Explore, comparing the total number of environment interactions in tasks of varying complexity in a partially observable environment. The mean and standard deviation are calculated over 3 runs for each algorithm. A maximum of two million interactions depicts that the algorithm failed to complete the task.

the time taken for the agent to learn to complete a task, there is still a draw back in planning time.

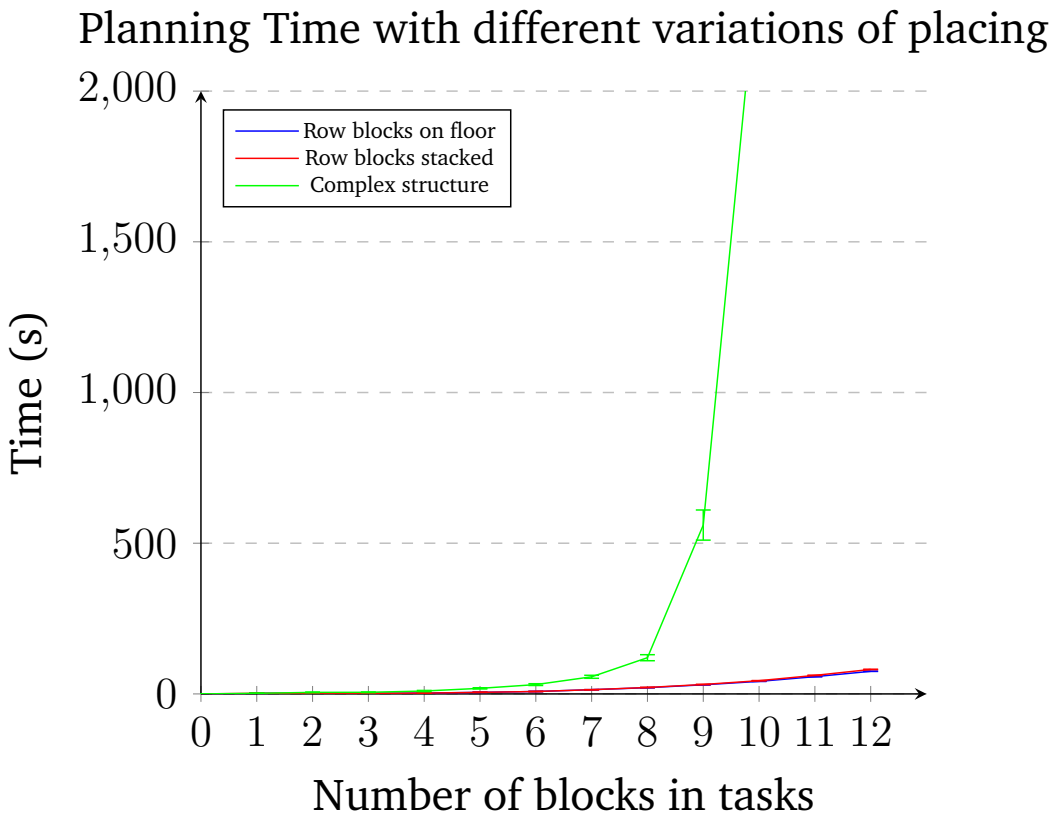


Figure 6.2: A time comparison between different variants of structure shape for Plan-Explore, being a line of blocks that increase horizontally, a wall structure of vertical size two and a complex shape, with these results being an average over 3 runs. These structures can be viewed in Appendix A.

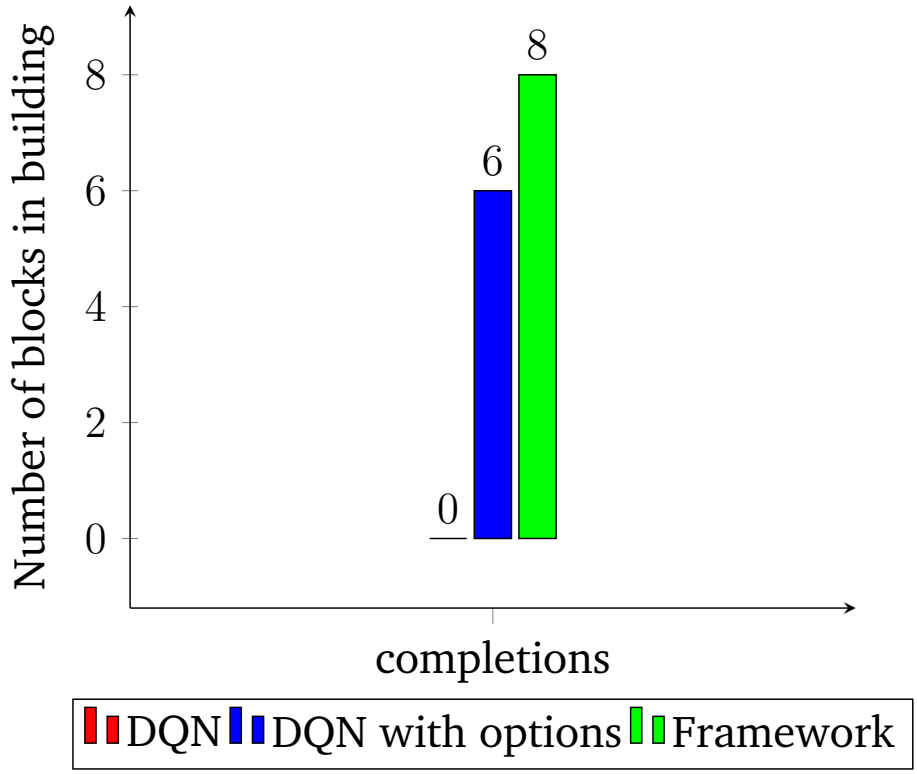


Figure 6.3: A difference in the maximum number of blocks each framework can place.

6.2 Overall Planning Time

Due to classical planning being vital to this research, we also test the performance of multiple search algorithms and heuristics to evaluate their suitability for solving complex planning tasks efficiently. Classical planning, which focuses on finding sequences of actions to achieve a specified goal from a given initial state, benefits greatly from effective search strategies and heuristics. By assessing the planning times and stability of each algorithm-heuristic combination, we aim to identify those most capable of handling the specific challenges posed by our domain.

A heuristic is a method that helps guide the search algorithm towards a solution more efficiently by estimating the cost or distance to the goal. Here, we evaluate four heuristics:

- **FF (Fast Forward)**: uses a relaxed plan heuristic to estimate the cost of reaching the goal, achieving efficient planning by ignoring some action preconditions to simplify the search space [Hoffmann and Nebel 2001]. This design allows FF to quickly move toward the goal, especially in large or complex environments."
- **Add (Additive)**: calculates the estimated cost of reaching the goal state by summing the heuristic estimates for each individual subgoal. This approach assumes that the cost of achieving each subgoal is independent, which simplifies the planning process and can lead to effective solutions in problems with clear subgoal dependencies.
- **CG (Cost Graph)**: is a cost-based heuristic that uses causal graph analysis to generate detailed estimates of future costs. By analyzing the relationships between actions and their effects, it provides more informed cost estimates, which can improve the efficiency of the planning process in complex environments.
- **Blind**: is a basic heuristic that lacks any knowledge of the environment, resulting in random exploration without any guidance or prioritization of actions. This approach does not use any domain-specific information and relies solely on brute-force exploration..

Given the variety of available heuristics, we can analyze how different search strategy performs when paired with different heuristics. The search strategies chosen to test are:

- **A* (A-Star)**: A search algorithm that combines path cost and heuristic estimates to find optimal solutions by evaluating the total cost of each potential path. It balances the actual cost of reaching a node and the estimated cost to the goal, ensuring an efficient search for the best solution.
- **E Greedy**: A search algorithm that balances exploration and exploitation by selecting actions that either maximize immediate rewards (exploitation) or explore new options (exploration) with a certain probability. This balance allows for both refining known strategies and discovering new possibilities.

- **Lazy Greedy**: A variant of the greedy search algorithm that prioritizes actions based on immediate perceived promise, focusing on the most promising actions without considering the entire search space at each step. This approach is efficient in finding good solutions quickly, though not necessarily optimal.
- **EHC (Enforced Hill Climbing)**: An iterative search algorithm that climbs towards a goal by exploring paths with better cost estimates, repeatedly selecting the path with the most favourable cost until a local optimum or goal is reached. It incorporates enforced backtracking when stuck in local minima, promoting further exploration.

Viewing the comparison Table 6.1 and Table 6.2 , it is evident that the best overall search algorithm for planning in our domain is A* with the Add heuristic. All search algorithms with both the FF and Add heuristics show efficient timings, whereas the CG (Cost-based) and Blind heuristics exhibit significantly worse search times, especially for more complex tasks like arranging 5 blocks. This highlights that, while all search strategies are technically viable, the choice of heuristic plays a critical role in determining efficiency.

	FF	Add	CG	Blind
A*	$\mathbf{1.458 \pm 0.0148}$	1.494 ± 0.0114	2.958 ± 0.0130	1.848 ± 0.0045
E Greedy	1.464 ± 0.0152	1.504 ± 0.0114	4.984 ± 0.0195	1.794 ± 0.0055
Lazy Greedy	1.460 ± 0.0122	1.506 ± 0.0089	3.794 ± 0.0089	1.966 ± 0.0089
EHC	1.922 ± 0.0164	1.922 ± 0.0110	1.906 ± 0.0152	1.986 ± 0.0167

Table 6.1: Comparison of search algorithms and heuristics for planning the collection and placement of 3 blocks in a row, with results calculated in seconds, averaged over 5 samples per algorithm and heuristic. The $\pm$ values represent the standard deviations across the 5 samples for each algorithm and heuristic.

The results also reveal that A* consistently performs well across tasks, with low variance and stable timings, making it a reliable choice when paired with FF or Add. In contrast, Lazy Greedy and E Greedy search algorithms show high planning times with the CG heuristic, particularly in larger tasks, suggesting that CG may not be well-suited for more complex problem instances. Notably, Enforced Hill Climbing (EHC) could not produce a solution with the CG heuristic when dealing with 5 blocks, indicating a limitation of EHC when combined with certain cost-based heuristics.

	FF	Add	CG	Blind
A*	25.79 ± 0.008	$\mathbf{22.34 \pm 0.08}$	1930.97 ± 6.48	2337.15 ± 7.37
E Greedy	25.66 ± 0.07	25.21 ± 0.11	5473.24 ± 23.96	2138.53 ± 14.57
Lazy Greedy	25.546 ± 0.04	25.266 ± 0.04	10196.46 ± 24.88	2807.11 ± 21.09
EHC	25.54 ± 0.03	25.28 ± 0.02	NA	2782.52 ± 42.38

Table 6.2: Comparison of search algorithms and heuristics for planning the collection and placement of 5 blocks in a row, with results calculated in seconds, averaged over 5 samples per algorithm and heuristic. The $\pm$ values represent the standard deviations across the 5 samples for each algorithm and heuristic.

Additionally, there is a marked increase in planning times for all algorithms as task complexity increases (from 3 blocks to 5 blocks), with CG showing the most significant impact. This further emphasizes that for larger or more complex problems, heuristic selection becomes even more critical to ensure efficiency in planning times.

Chapter 7

Conclusion

Completing complex tasks in a 3D environment has been shown to be a particularly challenging class of problem due to the sparse reward signals and the partial observability of these environments. We have reviewed how this is eased by the use of options to aid reinforcement learning, and that while classical planning is capable of completing tasks, it requires a fully observable state space to do so.

We, thus, introduce our framework formally known as Plan-Explore, in which we leverage the convenient outcome of planning methods to plan and execute complex tasks that can be achieved in a complex environment with a partially observable state space. Plan-Explore bridges the gap between planning in a symbolic and pixel domain and achieving goals within a partially observable domain without the limitations of the sparse reward problem and partial observability.

Through the use of Minecraft as an example of a partially observable state space we have demonstrated that our framework well exceeds the capabilities of reinforcement learning methods such as a basic DQN or a DQN with options. This is done by testing these models on Minecraft tasks with increasing complexity by expanding the size of the build until the basic DQN algorithm as well as the more advanced DQN aided by options reached a complexity limit at which they could no longer compete with the Plan-Explore framework. Plan-Explore outperforms both these baseline models in the number of environment interactions taken to complete a goal. The contributions our framework provides formalises a method in which we can reliably handle partially observable environments and continually explore within them in order to complete complex tasks. We also present a method that enables us to identify and extract a specific object from the goal specification when the planner encounters a failure. This approach allows us to isolate the problematic object and facilitates its collection within the pixel domain, thereby enabling a revised plan that avoids failure due to unmet object requirements.

We conclude that, while planning is essential for achieving complex tasks, our approach exhibits exponential time growth as state complexity increases. Although our solution is achievable with the selected planner, completing highly complex tasks efficiently may

require more specialized planners. To handle this task type effectively, planners will need further refinement and targeted development.

Our framework Plan-Explore successfully outperforms conventional reinforcement learning in handling complex tasks in a partially observable infinite state space.

Future works include confronting the planning time taken for increasing task complexities, as current planning has an exponential growth of planning time with an increasing complexity. Other future works include formalising the Minecraft domain in other planning frameworks as well as formalising a symbolic Minecraft environment capable of handling a large number of block and item types.

The Plan-Explore framework opens new avenues in planning as it is capable of achieving complex tasks in unpredictable environments and has potential impacts in the field of robotics with applications in construction and engineering .

Appendix A

Planning Test Structures

A.1 Line of Blocks

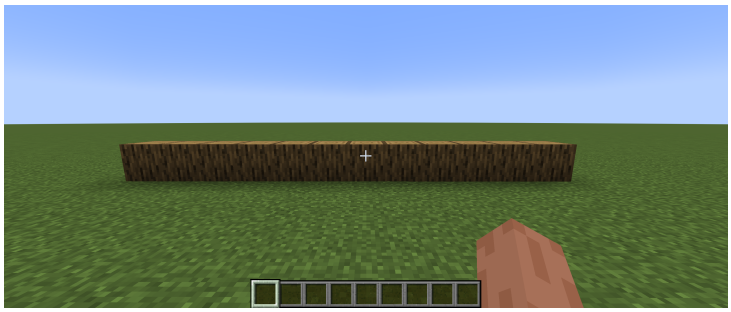


Figure A.1: A representation of the planning goal of placing blocks in a row.

A.2 Stacked Blocks



Figure A.2: A representation of the planning goal for stacking blocks in a row

A.3 Complex Structure



Figure A.3: A representation of the planning goal for a complex structure

Appendix B

PDDL

B.1 Planning Operators

B.1.1 move-north

```
     (:action move-north
    :parameters (?ag - agent ?x - position ?y - position ?
       z_start - position ?z_end - position)
    :precondition (and
        (agent-alive ?ag)
        (agent-on-floor ?ag)
        (at-x ?ag ?x)
        (at-y ?ag ?y)
        (at-z ?ag ?z_start)
        (are-seq ?z_start ?z_end)
        (not (exists (?bl - block) (and
            (block-present ?bl)
            (at-x ?bl ?x)
            (at-y ?bl ?y)
            (at-z ?bl ?z_end)
        )))
    )
    :effect (and
        (not (at-z ?ag ?z_start))
        (at-z ?ag ?z_end)
    )
)
```

B.1.2 move-south

```
   (:action move-south
    :parameters (?ag - agent ?x - position ?y - position ?
       z_start - position ?z_end - position)
```

```
    :precondition (and
        (agent-alive ?ag)
        (agent-on-floor ?ag)
        (at-x ?ag ?x)
        (at-y ?ag ?y)
        (at-z ?ag ?z_start)
        (are-seq ?z_end ?z_start)
        (not (exists (?bl - block) (and
            (block-present ?bl)
            (at-x ?bl ?x)
            (at-y ?bl ?y)
            (at-z ?bl ?z_end)
        )))
    )
    :effect (and
        (not (at-y ?ag ?y_start))
        (at-y ?ag ?y_end)
    )
)
```

B.1.3 move-east

```
    (:action move-east
    :parameters (?ag - agent ?x_start - position ?x_end -
       position ?y - position ?z - position)
    :precondition (and
        (agent-alive ?ag)
        (agent-on-floor ?ag)
        (at-x ?ag ?x_start)
        (at-y ?ag ?y)
        (at-z ?ag ?z)
        (are-seq ?x_start ?x_end)
        (not (exists (?bl - block) (and
            (block-present ?bl)
            (at-x ?bl ?x_end)
            (at-y ?bl ?y)
            (at-z ?bl ?z)
        )))
    )
    :effect (and
        (not (at-x ?ag ?x_start))
        (at-x ?ag ?x_end)
    )
)
```

B.1.4 move-west

```
    (:action move-west
    :parameters (?ag - agent ?x_start - position ?x_end -
       position ?y - position ?z - position)
    :precondition (and
        (agent-alive ?ag)
        (agent-on-floor ?ag)
        (at-x ?ag ?x_start)
        (at-y ?ag ?y)
        (at-z ?ag ?z)
        (are-seq ?x_end ?x_start)
        (not (exists (?bl - block) (and
            (block-present ?bl)
            (at-x ?bl ?x_end)
            (at-y ?bl ?y)
            (at-z ?bl ?z)
        )))
    )
    :effect (and
        ; (decrease (x ?ag) 1)
        (not (at-x ?ag ?x_start))
        (at-x ?ag ?x_end)
    )
)
)
```

B.1.5 jump-north-onto-block

```
    (:action jump-north-onto-block
    :parameters (?bl - block ?ag - agent ?x - position ?
       y_start - position ?y_end - position ?z_start -
       position ?z_end - position)
    :precondition (and
        (agent-alive ?ag)
        (are-seq ?y_start ?y_end)
        (are-seq ?z_start ?z_end)

        (at-x ?ag ?x)
        (at-y ?ag ?y_start)
        (at-z ?ag ?z_start)

        (exists (?bl - block) (and
            (block-present ?bl)
            (at-x ?bl ?x)
            (at-y ?bl ?y_end)
            (at-z ?bl ?z_start)
        ))
    )
```

```
    :effect (and
        (not (agent-on-floor ?ag))
        (not (at-y ?ag ?y_start))
        (not (at-z ?ag ?z_start))
        (at-y ?ag ?y_end)
        (at-z ?ag ?z_end)
    )
)
```

B.1.6 jump-south-onto-block

```
    (:action jump-south-onto-block
    :parameters (?ag - agent ?x - position ?y_start -
       position ?y_end - position ?z_start - position ?z_end
       - position)
    :precondition (and
        (agent-alive ?ag)
        (are-seq ?y_end ?y_start)
        (are-seq ?z_start ?z_end)

        (at-x ?ag ?x)
        (at-y ?ag ?y_start)
        (at-z ?ag ?z_start)

        (exists (?bl - block) (and
            (block-present ?bl)
            (at-x ?bl ?x)
            (at-y ?bl ?y_end)
            (at-z ?bl ?z_start)
        ))
    )
    :effect (and
        (not (agent-on-floor ?ag))
        (not (at-y ?ag ?y_start))
        (not (at-z ?ag ?z_start))
        (at-y ?ag ?y_end)
        (at-z ?ag ?z_end)
    )
)
```

B.1.7 jump-east-onto-block

```
   (:action jump-east-onto-block
    :parameters (?ag - agent ?x_start - position ?x_end -
       position ?y - position ?z_start - position ?z_end -
       position)
    :precondition (and
```

```
        (agent-alive ?ag)

        (are-seq ?x_start ?x_end)
        (are-seq ?z_start ?z_end)

        (at-x ?ag ?x_start)
        (at-y ?ag ?y)
        (at-z ?ag ?z_start)

        (exists (?bl - block) (and
            (block-present ?bl)
            (at-x ?bl ?x_end)
            (at-y ?bl ?y)
            (at-z ?bl ?z_start)
        ))
    )
    :effect (and
        (not (agent-on-floor ?ag))
        (not (at-x ?ag ?x_start))
        (not (at-z ?ag ?z_start))
        (at-x ?ag ?x_end)
        (at-z ?ag ?z_end)
    )
)
```

B.1.8 jump-west-onto-block

```
   (:action jump-west-onto-block
    :parameters (?ag - agent ?x_start - position ?x_end -
       position ?y - position ?z_start - position ?z_end -
       position)
    :precondition (and
        (agent-alive ?ag)

        (are-seq ?x_end ?x_start)
        (are-seq ?z_start ?z_end)

        (at-x ?ag ?x_start)
        (at-y ?ag ?y)
        (at-z ?ag ?z_start)

        (exists (?bl - block) (and
            (block-present ?bl)
            (at-x ?bl ?x_end)
            (at-y ?bl ?y)
            (at-z ?bl ?z_start)
        ))
```

```
    )
    :effect (and
        (not (agent-on-floor ?ag))
        (not (at-x ?ag ?x_start))
        (not (at-z ?ag ?z_start))
        (at-x ?ag ?x_end)
        (at-z ?ag ?z_end)
    )
)
```

B.1.9 move-north-on-block

```
    (:action move-north-on-block
    :parameters (?ag - agent ?x - position ?y_start -
       position ?y_end - position ?z - position ?z_below -
       position)
    :precondition (and
        (agent-alive ?ag)
        (not (agent-on-floor ?ag))
        (at-x ?ag ?x)
        (at-y ?ag ?y)
        (at-z ?ag ?z_start)
        (are-seq ?y_below ?y)
        (are-seq ?z_start ?z_end)
        (exists (?bl - block) (and
            (block-present ?bl)
            (at-x ?bl ?x)
            (at-y ?bl ?y_below)
            (at-z ?bl ?z_end)
        ))
    )
    :effect (and
        (not (at-y ?ag ?z_start))
        (at-y ?ag ?z_end)
    )
)
```

B.1.10 move-south-on-block

```
    (:action move-south-on-block
    :parameters (?ag - agent ?x - position ?y - position ?
       y_below - position ?z_start - position ?z_end -
       position)
    :precondition (and
        (agent-alive ?ag)
        (not (agent-on-floor ?ag))
        (at-x ?ag ?x)
```

```
        (at-y ?ag ?y)
        (at-z ?ag ?z_start)
        (are-seq ?y_below ?y)
        (are-seq ?z_end ?z_start)
        (exists (?bl - block) (and
            (block-present ?bl)
            (at-x ?bl ?x)
            (at-y ?bl ?y_below)
            (at-z ?bl ?z_end)
        ))
    )
    :effect (and
        (not (at-y ?ag ?z_start))
        (at-y ?ag ?z_end)
    )
)
```

B.1.11 move-east-on-block

```
    (:action move-east-on-block
    :parameters (?ag - agent ?x_start - position ?x_end -
       position ?y - position ?z - position ?z_below -
       position)
    :precondition (and
        (agent-alive ?ag)
        (not (agent-on-floor ?ag))
        (at-x ?ag ?x_start)
        (at-y ?ag ?y)
        (at-z ?ag ?z)
        (are-seq ?x_start ?x_end)
        (are-seq ?z_below ?z)
        (exists (?bl - block) (and
            (block-present ?bl)
            (at-x ?bl ?x_end)
            (at-y ?bl ?y)
            (at-z ?bl ?z_below)
        ))
    )
    :effect (and
        (not (at-x ?ag ?x_start))
        (at-x ?ag ?x_end)
    )
)
```

B.1.12 move-west-on-block

```
    (:action move-west-on-block
```

```
    :parameters (?ag - agent ?x_start - position ?x_end -
       position ?y - position ?z - position ?z_below -
       position)
    :precondition (and
        (agent-alive ?ag)
        (not (agent-on-floor ?ag))
        (at-x ?ag ?x_start)
        (at-y ?ag ?y)
        (at-z ?ag ?z)
        (are-seq ?x_end ?x_start)
        (are-seq ?z_below ?z)
        (exists (?bl - block) (and
            (block-present ?bl)
            (at-x ?bl ?x_end)
            (at-y ?bl ?y)
            (at-z ?bl ?z_below)
        ))
    )
    :effect (and
        ; (decrease (x ?ag) 1)
        (not (at-x ?ag ?x_start))
        (at-x ?ag ?x_end)
    )
)
```

B.1.13 destroy-wooden-block

```
    (:action destroy-wooden-block
    :parameters (
        ?ag - agent
        ?b - wooden-block
        ?x - position
        ?y_ag - position
        ?y_bl - position
        ?z - position
        ?n_start - count
        ?n_end - count
    )
    :precondition (and
        (at-x ?ag ?x)
        (at-x ?b ?x)
        (at-y ?ag ?y_ag)
        (at-y ?b ?y_bl)
        (at-z ?ag ?z)
        (at-z ?b ?z)
        (block-present ?b)
        (agent-has-n-wooden-blocks ?ag ?n_start)
```

```
        (are-seq ?n_start ?n_end)
        (are-seq ?y_ag ?y_bl)
    )
    :effect (and
        (not (block-present ?b))
        (not (at-x ?b ?x))
        (not (at-y ?b ?y_bl))
        (not (at-z ?b ?z))

        (not (agent-has-n-wooden-blocks ?ag ?n_start))
        (agent-has-n-wooden-blocks ?ag ?n_end)
        (not (on-floor ?b))
    )
)
```

B.1.14 place-wooden-block-on-floor

```
    (:action place-wooden-block-on-floor
    :parameters (
        ?ag - agent
        ?b - wooden-block
        ?n_start - count
        ?n_end - count
        ?x - position
        ?y_start - position
        ?y_end - position
        ?z - position

    )
    :precondition (and
        (are-seq ?y_start ?y_end)
        (are-seq ?n_end ?n_start)
        (agent-has-n-wooden-blocks ?ag ?n_start)
        (at-x ?ag ?x)
        (at-y ?ag ?y_start)
        (at-z ?ag ?z)
        (not (block-present ?b))
    )
    :effect (and
        (block-present ?b)
        (at-x ?b ?x)
        (at-y ?b ?y_end)
        (at-z ?b ?z)
        (not (agent-has-n-wooden-blocks ?ag ?n_start))
        (agent-has-n-wooden-blocks ?ag ?n_end)
        (on-floor ?b)
    )
```

```
)
```

B.1.15 place-wooden-block-on-block

```
    (:action place-wooden-block-on-block
        :parameters (
            ?ag - agent
            ?b - wooden-block
            ?n_start - count
            ?n_end - count
            ?x - position
            ?y_start - position
            ?y_end - position
            ?z - position
            ?z_above - position
            ?b_below - block
        )
        :precondition (and
            ; (>= (agent-num-wooden-block ?ag) 1)
            (agent-has-n-wooden-blocks ?ag ?n_start)
            (are-seq ?n_end ?n_start)
            (at-x ?ag ?x)
            (at-y ?ag ?y_start)
            (are-seq ?y_start ?y_end) ;; values increase from
                south to north
            (at-z ?ag ?z)
            (not (block-present ?b))

            (block-present ?b_below) ; Ensure there is
               another block ?other on which we can place the
                block ?b
            (on-floor ?b_below) ; Ensure the other block ?
               other is on the floor
            (at-x ?b_below ?x)
            (at-y ?b_below ?y_end)
            (at-z ?b_below ?z)
            (are-seq ?z ?z_above)

        )
        :effect (and
            (block-present ?b)
            (at-x ?b ?x)
            (at-y ?b ?y_end)
            (at-z ?b ?z_above)
            (not (agent-has-n-wooden-blocks ?ag ?n_start))
            (agent-has-n-wooden-blocks ?ag ?n_end)
```

```
            (on-top-of ?b ?b_below) ; Effect to indicate the
               block ?b is placed on top of block ?other
        )
    )
```

References

[Aeronautiques *et al.* 1998] Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins SRI, Anthony Barrett, Dave Christianson, et al. Pddl— the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

[Aluru *et al.* 2015] Krishna Chaitanya Aluru, Stefanie Tellex, John Oberlin, and James MacGlashan. Minecraft as an experimental world for ai in robotics. In *2015 aaai fall symposium series*, 2015.

[Arjona-Medina *et al.* 2019] Jose A. Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. *RUDDER: Return Decomposition for Delayed Rewards*, 2019.

[Asai and Fukunaga 2015] Masataro Asai and Alex Fukunaga. Solving large-scale planning problems by decomposition and macro generation. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, pages 16–24, 2015.

[Barto and Mahadevan 2003] Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1):41–77, 2003.

[Bellman 1957] RE Bellman. A markov decision process. journal of mathematical mechanics. 1957.

[Benyamin *et al.* 2023] Yarin Benyamin, Argaman Mordoch, Shahaf S Shperberg, and Roni Stern. Model learning to solve minecraft tasks. In *PRL Workshop Series {\textendash} Bridging the Gap Between AI Planning and Reinforcement Learning*, 2023.

[Bercher *et al.* 2014] Pascal Bercher, Shawn Keen, and Susanne Biundo. Hybrid planning heuristics based on task decomposition graphs. In *Seventh Annual Symposium on Combinatorial Search*, 2014.

[Boutilier *et al.* 1999] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[Chevalier-Boisvert *et al.* 2023] Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo Perez-Vicente, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel

Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. In *Advances in Neural Information Processing Systems 36, New Orleans, LA, USA*, December 2023.

[Christoffersen *et al.* 2023] Phillip JK Christoffersen, Andrew C Li, Rodrigo Toro Icarte, and Sheila A McIlraith. Learning symbolic representations for reinforcement learning of non-markovian behavior. *arXiv preprint arXiv:2301.02952*, 2023.

[Ecoffet *et al.* 2019] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*, 2019.

[Fan *et al.* 2022] Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *Advances in Neural Information Processing Systems*, 35:18343–18362, 2022.

[Frazier and Riedl 2019] Spencer Frazier and Mark Riedl. Improving deep reinforcement learning in minecraft with action advice. In *Proceedings of the AAAI conference on artificial intelligence and interactive digital entertainment*, volume 15, pages 146–152, 2019.

[Garnelo *et al.* 2016] Marta Garnelo, Kai Arulkumaran, and Murray Shanahan. Towards deep symbolic reinforcement learning. *arXiv preprint arXiv:1609.05518*, 2016.

[Guan *et al.* 2022] Lin Guan, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging approximate symbolic models for reinforcement learning via skill diversity. In *International Conference on Machine Learning*, pages 7949–7967. PMLR, 2022.

[Guss *et al.* 2019] William H Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. Minerl: A large-scale dataset of minecraft demonstrations. *arXiv preprint arXiv:1907.13440*, 2019.

[Guss *et al.* 2020] William H. Guss, Mario Ynocente Castro*, Sam Devlin*, Brandon Houghton*, Noboru Sean Kuno*, Crissman Loomis*, Stephanie Milani*, Sharada Mohanty*, Keisuke Nakata*, Ruslan Salakhutdinov*, John Schulman*, Shinya Shiroshita*, Nicholay Topin*, Avinash Ummadisingu*, and Oriol Vinyals*. Neurips 2020 competition: The MineRL competition on sample efficient reinforcement learning using human priors. *NeurIPS Competition Track*, 2020.

[Hart *et al.* 1968] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Helmert *et al.* 2008] Malte Helmert, Gabriele Röger, et al. How good is almost perfect?. In *AAAI*, volume 8, pages 944–949, 2008.

[Helmert 2006] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[Hill *et al.* 2023] William Hill, Ireton Liu, Anita De Mello Koch, Damion Harvey, George Konidaris, and Steven James. Mineplanner: A benchmark for long-horizon planning in large minecraft worlds. *arXiv preprint arXiv:2312.12891*, 2023.

[Hoffmann and Nebel 2001] Jörg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[Howard 1960] Ronald Howard. *Dynamic programming and Markov processes*. John Wiley, 1960.

[Hussein *et al.* 2017] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

[Icarte *et al.* 2022] Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano, and Sheila A McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research*, 73:173–208, 2022.

[Illanes *et al.* 2020] León Illanes, Xi Yan, Rodrigo Toro Icarte, and Sheila A McIlraith. Symbolic plans as high-level instructions for reinforcement learning. In *Proceedings of the international conference on automated planning and scheduling*, volume 30, pages 540–550, 2020.

[James *et al.* 2020a] Steven James, Benjamin Rosman, and George Konidaris. Learning object-centric representations for high-level planning in minecraft. In *Proceedings of Workshop on Object-Oriented Learning at ICML*, 2020.

[James *et al.* 2020b] Steven James, Benjamin Rosman, and George Konidaris. Learning portable representations for high-level planning. In *International Conference on Machine Learning*, pages 4682–4691. PMLR, 2020.

[Jonsson and Lotinac 2015] Anders Jonsson and Damir Lotinac. Automatic generation of htns from pddl. *Planning and Learning (PAL-15)*, page 15, 2015.

[Kaelbling *et al.* 1998] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[Kambhampati and Srivastava 1996] Subbarao Kambhampati and Biplav Srivastava. Unifying classical planning approaches. *Arizona State University ASU CSE TR*, pages 96–006, 1996.

[Kanervisto *et al.* 2020] Anssi Kanervisto, Janne Karttunen, and Ville Hautamäki. Playing minecraft with behavioural cloning. In *NeurIPS 2019 Competition and Demonstration Track*, pages 56–66. PMLR, 2020.

[Konidaris and Barto 2007] George Dimitri Konidaris and Andrew G Barto. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, pages 895–900, 2007.

[Konidaris and Barto 2009] George Konidaris and Andrew Barto. Skill chaining: Skill discovery in continuous domains. In *the Multidisciplinary Symposium on Reinforcement Learning, Montreal, Canada*, 2009.

[Konidaris *et al.* 2014] George Konidaris, Leslie Kaelbling, and Tomas Lozano-Perez. Constructing symbolic representations for high-level planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[Konidaris *et al.* 2018] George Konidaris, Leslie Pack Kaelbling, and Tomas Lozano-Perez. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, 61:215–289, 2018.

[Leonetti *et al.* 2016] Matteo Leonetti, Luca Iocchi, and Peter Stone. A synthesis of automated planning and reinforcement learning for efficient, robust decision-making. *Artificial Intelligence*, 241:103–130, 2016.

[Lin *et al.* 2021] Zichuan Lin, Junyou Li, Jianing Shi, Deheng Ye, Qiang Fu, and Wei Yang. Juewu-mc: Playing minecraft with sample-efficient hierarchical reinforcement learning. *arXiv preprint arXiv:2112.04907*, 2021.

[Lipovetzky and Geffner 2017] Nir Lipovetzky and Hector Geffner. Best-first width search: Exploration and exploitation in classical planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[Lyu *et al.* 2019] Daoming Lyu, Fangkai Yang, Bo Liu, and Steven Gustafson. Sdrl: interpretable and data-efficient deep reinforcement learning leveraging symbolic planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 2970–2977, 2019.

[McDermott *et al.* 1998] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language. 1998.

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

[Nau *et al.* 2003] Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. Shop2: An htn planning system. *Journal of artificial intelligence research*, 20:379–404, 2003.

[Nayyar *et al.* 2023] Rashmeet Kaur Nayyar, Shivanshu Verma, and Siddharth Srivastava. Learning generalizable symbolic options for transfer in reinforcement learning. In *NeurIPS 2023 Workshop on Generalization in Planning*, 2023.

[Pathak *et al.* 2017] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, pages 2778–2787. PMLR, 2017.

[Perkins *et al.* 1999] Theodore J Perkins, Doina Precup, et al. *Using options for knowledge transfer in reinforcement learning*. Technical report, Citeseer, 1999.

[Pineau *et al.* 2003] Joelle Pineau, Geoff Gordon, Sebastian Thrun, et al. Point-based value iteration: An anytime algorithm for pomdps. In *Ijcai*, volume 3, pages 1025–1032, 2003.

[Russell and Norvig 2016] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 2016.

[Shah *et al.* 2021] Rohin Shah, Cody Wild, Steven H Wang, Neel Alex, Brandon Houghton, William Guss, Sharada Mohanty, Anssi Kanervisto, Stephanie Milani, Nicholay Topin, et al. The minerl basalt competition on learning from human feedback. *arXiv preprint arXiv:2107.01969*, 2021.

[Silver and Veness 2010] David Silver and Joel Veness. Monte-carlo planning in large pomdps. *Advances in neural information processing systems*, 23, 2010.

[Stolle and Precup 2002] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer, 2002.

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

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

[Thomaz *et al.* 2005] Andrea Lockerd Thomaz, Guy Hoffman, and Cynthia Breazeal. Real-time interactive reinforcement learning for robots. In *AAAI 2005 workshop on human comprehensible machine learning*, pages 9–13, 2005.

[Thrun 2000] Sebastian Thrun. Probabilistic algorithms in robotics. *Ai Magazine*, 21(4):93–93, 2000.

[Torabi *et al.* 2018] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. 2018.

[Turing 2009] Alan M Turing. *Computing machinery and intelligence*. Springer, 2009.

[Ugur and Piater 2015a] Emre Ugur and Justus Piater. Bottom-up learning of object categories, action effects and logical rules: From continuous manipulative exploration to symbolic planning. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2627–2633. IEEE, 2015.

[Ugur and Piater 2015b] Emre Ugur and Justus Piater. Refining discovered symbols with multi-step interaction experience. In *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, pages 1007–1012. IEEE, 2015.

[Wang *et al.* 2009] Zhenchao Wang, Haibin Duan, and Xiangyin Zhang. An improved greedy genetic algorithm for solving travelling salesman problem. In *2009 Fifth International Conference on Natural Computation*, volume 5, pages 374–378. IEEE, 2009.

[Wang *et al.* 2018] William Yang Wang, Jiwei Li, and Xiaodong He. Deep reinforcement learning for NLP. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts*, pages 19–21, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[Watkins 1989] Christopher Watkins. *Learning from delayed rewards*. PhD thesis, Cambridge University, 1989.

[Wichlacz *et al.* 2019] Julia Wichlacz, Alvaro Torralba, and Jörg Hoffmann. Construction-planning models in minecraft. 2019.

[Yang *et al.* 2018] Fangkai Yang, Daoming Lyu, Bo Liu, and Steven Gustafson. Peorl: Integrating symbolic planning and hierarchical reinforcement learning for robust decision-making. *arXiv preprint arXiv:1804.07779*, 2018.