Chair of Computer Science 6 (Machine Learning and Reasoning)

RWTH AACHEN UNIVERSITY

The present work was submitted to the Chair of Machine Learning and Reasoning.
Diese Arbeit wurde vorgelegt am Lehrstuhl für Maschinelles Lernen und Inferenz.

Exploring the Limits of Relational Graph Neural Networks in PushWorld

Erkundung der Grenzen von Relational Graph Neural Networks in PushWorld

Bachelor Thesis
Bachelorarbeit

Presented by / Vorgelegt von

Samuel Stante
434481

Supervised by / Betreut von Michael Aichmüller, M.Sc.

1st Examiner / 1. Prüfer Prof. Hector Geffner, Ph.D.

2nd Examiner / 2. Prüfer Prof. Dr. rer. nat. Christopher Morris

Aachen, September 24, 2024

Abstract

Prior work of Ståhlberg *et al.* utilizes reinforcement learning to learn policies on classical planning problems, using a policy gradient actor-critic algorithm with a message-passing graph neural network as a function approximator. While their approach performed well across various domains, it faces potential limitations due to the expressivity of graph neural networks, which lies between GC_2 and C_2 logic. One can add derived predicates to the domain to overcome this, enabling the graph neural network to express additional features required by policies.

This thesis explores the graph neural network's expressivity limitation in the PushWorld domain within the approach of Ståhlberg *et al.* PushWorld is a complex, two-dimensional grid-based environment that combines several challenges. Compared to the domains used by Ståhlberg *et al.*, PushWorld is more complex, has larger state spaces, and introduces unseen challenges. It is similar to the well-known Sokoban domain. Taking advantage of this similarity, we introduce a simplified version of PushWorld, RestrictedPushWorld. We sketch an idea of a possible proof that determining if RestrictedPushWorld problems are solvable is NP-hard by illustrating a reduction to a known NP-hard problem. While this hardness implies the absence of a general policy, we can still strive to obtain policies that solve as many problems as possible.

Besides necessary modifications to the code of Ståhlberg *et al.*, we implement an incomplete yet correct dead-end heuristic for RestrictedPushWorld used during training. Based on the tests that compare models trained with and without it, this heuristic does not necessarily help avoid dead-ends. In fact, models trained with the heuristic performed worse in other assessments, having issues such as aimless wandering and random box pushing, even in puzzles with small grid sizes.

The test coverages we achieved surpassed those from the reinforcement learning approaches by the PushWorld authors Kansky *et al.* We also conducted tests that evaluated the models' ability to systematically find and clear paths blocked by boxes. Although the models solved some problems, their solutions lacked a consistent system and appeared to be found by chance. Despite efforts, it was challenging to pinpoint precise reasons for the observed behavior, especially finding limitations regarding the expressivity of graph neural networks.

Contents

1 Introduction **1**

2 Background **3**

2.1 Classical Planning . . . 3

2.1.1 STRIPS and PDDL . . . 4

2.1.2 Generalized Planning . . . 5

2.2 Reinforcement Learning . . . 5

2.2.1 Markov Decision Processes . . . 5

2.2.2 Returns, Episodes, and Discounting . . . 6

2.2.3 Policies, Value Functions, and Bellman Equations . . . 7

2.2.4 Algorithm Categories . . . 9

2.2.5 Dynamic Programming . . . 10

2.2.6 Approximating Value Functions . . . 11

2.2.7 Policy Gradient Methods . . . 12

2.2.8 Advantages and Limitations . . . 13

2.3 Artificial Neural Networks . . . 13

2.3.1 Multilayer Perceptrons . . . 14

2.3.2 Convolutional Neural Networks . . . 14

2.3.3 Graph Neural Networks . . . 15

2.4 P vs. NP . . . 17

3 Related Work **18**

3.1 Learning General Policies with Policy Gradient Methods . . . 18

3.1.1 The Two Actor-Critic Algorithms . . . 18

3.1.2 Constructing the Value and Policy Function . . . 20

3.2 PushWorld: A Benchmark for Manipulation Planning . . . 21

3.2.1 Challenges of the Environment . . . 22

3.2.2 PushWorld PDDL . . . 23

3.2.3 State of the Art with Planning . . . 24

3.2.4 State of the Art with Reinforcement Learning . . . 25

3.3 Sokoban . . . 26

3.3.1 Sokoban PDDL . . . 26

4 Methods **27**
4.1 Adapting the Existing Sokoban PDDL . . . 27
4.2 Introducing the Variants . . . 28
4.3 Hardness of the Problem . . . 29
4.3.1 Monotone Linked Planar 3-SAT . . . 30
4.3.2 Modules . . . 31
4.3.3 Reduction Sketch . . . 32
4.3.4 Removing Walls . . . 34
4.4 Adapting the Existing Implementation . . . 34
4.4.1 Replay Buffer . . . 34
4.4.2 Validation . . . 35
4.4.3 Dead-End Heuristic . . . 36
4.4.4 Confident Actor But Incorrect Value Function . . . 38

5 Experiments **39**
5.1 Setup and Data . . . 39
5.2 Results . . . 40
5.2.1 V1 Results . . . 43
5.2.2 V2 Results . . . 44
5.2.3 V3 Results . . . 45
5.2.4 PushWorld Variants Results . . . 46
5.3 Special Tests . . . 47
5.3.1 Increasing Puzzle Sizes . . . 47
5.3.2 One-Step Dead-Ends (OSDEs) . . . 48
5.3.3 Corridor with Hooks . . . 50
5.3.4 Pillars . . . 51
5.3.5 Double Pillar . . . 53

6 Conclusion **54**

A Appendix **56**
A.1 PDDL Domains . . . 56
A.2 Collection of Challenging RestrictedPushWorld Puzzles . . . 60
A.3 Hyperparameters and Problem Sets . . . 61
A.4 Grid Size and Density Distributions . . . 62
A.5 Adjacent Three . . . 63

List of Acronyms **65**

List of Symbols **66**

List of Figures **69**

List of Tables **70**

List of Listings **71**

List of Algorithms **72**

List of References **73**

1 Introduction

Reinforcement learning [28, 30] is a machine learning paradigm where an agent learns through trial and error. The environment provides feedback through rewards or penalties, which the agent uses to adjust behavior, inspired by how humans learn from experience. Unlike supervised learning, where labeled data is necessary, in reinforcement learning, the agent explores the environment independently, making it suitable for various applications. However, reinforcement learning lacks a clear way to interpret learned behavior [28], as the obtained policy is often a black box and not formulated in formal language. This makes it hard to reason about it, leaving little room for correctness, robustness, and safety guarantees.

In contrast to reinforcement learning, planning [9, 28], another branch in artificial intelligence, expresses problems and policies in formal languages, such as the planning domain definition language, which allows for a precise description of the environment, its states, and actions. Thus, policies can often be proven to be correct and generalize over all possible problem instances of specific classes. The downside of planning is that the environment must be fully known and encoded in the formal language, which, in practice, must be done by hand. Also, planning methods tend to be computationally expensive.

Ståhlberg *et al.* [28] apply reinforcement learning methods to obtain general policies on classical planning problems. Specifically, they use a policy gradient actor-critic algorithm with a message-passing graph neural network, modified to accept relational structures as input, as the function approximator. Graph neural networks [26, 28] are known to generalize well over unseen data, are invariant to graph isomorphisms, and produce consistent output on graphs of any size once trained. In their evaluation across multiple domains, Ståhlberg *et al.* [28] report promising results in most, achieving 100% coverage in six out of ten domains, with near-optimal performance. Coverage in three other domains was around 70%-80%, while the last domain achieved only 36%.

The limited expressivity of graph neural networks may explain the suboptimal performance in some domains, as their expressivity falls between GC_2 and C_2 logic [2, 12, 16, 28]. C_2 extends first-order logic with counting quantifiers but limits the number of variables to two, while GC_2 represents the guarded fragment of C_2. To address this limitation, one can introduce derived predicates. In the domain with 36% coverage, Ståhlberg *et al.* [28] identified a feature that cannot be expressed in C_2 but requires at least C_3. Incorporating this feature as a derived predicate increased the coverage to 91%.

PushWorld by Kansky *et al.* [17] is another domain with a two-dimensional, grid-based environment where an agent must push boxes to their designated goal positions. The environment also includes obstacles, walls, and barriers, with all objects capable of having arbitrary shapes and sizes. PushWorld is specifically designed to be complex and to combine multiple challenges. It simulates natural human interactions with objects, pathfinding, and prioritization of multiple objectives in an abstract way. For instance, the agent may need to collect parts of a tool, assemble them, and use the tool to achieve a goal. PushWorld appears to be an extension of the well-known Sokoban domain with additional modifications.

This thesis aims to get a better understanding of the challenges of PushWorld in combination with the methods from Ståhlberg *et al.* [28] and to determine how to overcome them. Instead of PushWorld itself, we use a simplified version, RestrictedPushWorld, for two main reasons. First, the code of Ståhlberg *et al.* [28] does not support all features required by Kansky *et al.* [17], and implementing the necessary modifications is beyond the scope of this thesis. Second, RestrictedPushWorld makes it easier to analyze results and is already sufficient for exploring limitations of graph neural networks.

We introduce increasingly complex variants of RestrictedPushWorld that focus on specific scenarios. Additionally, we outline a possible proof that determining if a RestrictedPushWorld problem is solvable is generally NP-hard by sketching the idea of a reduction to a known NP-hard problem. This hardness implies that our approach has no general policy. Nevertheless, we can aim for policies that solve as many problems as possible. Furthermore, we also need to expand the existing code from Ståhlberg *et al.* [28]. The primary reason is that it generates complete state spaces for each training instance. While feasible for the problems they used due to their small enough state spaces, it becomes impractical for RestrictedPushWorld.

We address this by implementing a replay buffer, which stores only a limited number of states. In doing so, we also replace the old behavior of finding dead-ends through state spaces by introducing a correct but incomplete dead-end heuristic. We then compare models trained with and without this heuristic to determine their effectiveness in avoiding dead-ends. Additionally, we investigate how the performance of the models changes when the grid size or the number of normal boxes or walls is increased. Finally, we conduct a series of specialized tests to evaluate the models' ability to solve specific scenarios.

This thesis is structured as follows. Chapter 2 provides the relevant background knowledge of classical planning, reinforcement learning, and certain artificial neural networks used in this thesis. Chapter 3 reviews the related work of Ståhlberg *et al.* [28], and the environments PushWorld from Kansky *et al.* [17] and Sokoban. Chapter 4 describes the methods used for the experiments presented in Chapter 5. This includes modifications to the PDDL definitions, an introduction to the analyzed variants and their complexity, and the changes to the implementation of Ståhlberg *et al.* [28]. Finally, Chapter 6 concludes the thesis.

2 Background

This chapter provides an overview of the fundamental concepts and techniques relevant to the thesis. It begins with an introduction to two artificial intelligence (AI) areas: classical planning in Section 2.1 and reinforcement learning in Section 2.2, which Ståhlberg *et al.* [28] combine in their work. These sections build up crucial parts of their algorithms, which will be introduced in the next chapter. Since a graph neural network serves as a function approximator and multilayer perceptrons are used, for instance, in their readout functions, the chapter continues with a brief introduction to artificial neural networks in Section 2.3. This also includes convolutional neural networks, which were used for reinforcement learning by Kansky *et al.* [17] for PushWorld. Finally, it concludes with a refresher of the complexity classes P and NP in Section 2.4.

2.1 Classical Planning

Planning [9] is a branch of AI that involves generating a sequence of actions to achieve a specific goal from an initial state. Classical planning [9] is a subfield of planning in which the environment is deterministic and known. The following uses the notation and definitions of Geffner and Bonet [9] with slight modifications. They describe the model underlying classical planning as a state model $S = \langle S, s_0, S_G, A, f, c \rangle$ where

- S is a finite and discrete set of states,
- $s_0 \in S$ is the initial state,
- $S_G \subseteq S$ is a non-empty set of goal states,
- A is a finite and discrete set of actions, and $A(s) \subseteq A$ represents the set of actions that are applicable in a state $s \in S$,
- $f : A \times S \to S$ is the deterministic transition function, and
- $c : A \times S \to \mathbb{R}^+$ is the positive cost of performing an action in a state.

A solution or plan of this classical planning state model is a sequence of actions $a_0, \dots, a_n$ that lead from the initial state s_0 to a goal state s_{n+1} where $s_0, \dots, s_{n+1}$ is the state sequence generated by the actions. A plan is optimal if there is no other plan with a lower cost. Automated planners are used to automatically produce plans based on a compact representation of the classical planning state model [9].

2.1.1 STRIPS and PDDL

One such planner is the Stanford Research Institute Problem Solver (STRIPS) [7]. It introduces a well-defined language for classical planning, referred to likewise as STRIPS. Generally, when referring to STRIPS, we mean the language, not the planner. A problem in STRIPS is defined as $P = \langle F, O, I, G \rangle$ where

- F is the set of atoms,
- O is the set of actions,
- $I \subseteq F$ is the initial situation, and
- $G \subseteq F$ is the goal.

Each action $o \in O$ is described by an add-list $Add(o) \subseteq F$, a delete-list $Del(o) \subseteq F$, and a precondition-list $Pre(o) \subseteq F$. The add-list contains atoms that become true when applying the action. The delete-list contains atoms that become no longer true when applying the action. The precondition-list contains atoms that must be true to apply the action. STRIPS implicitly encodes the classical planning model in a compact form [9].

The planning domain definition language (PDDL) [1, 8, 11] builds on STRIPS and provides a standard syntax for problems, supporting variables, types, and more. A PDDL problem consists of a domain and an instance: $P = \langle I, D \rangle$. The domain describes the action schemas over predicates and consists of a set of predicate symbols and a set of action schemas. The actions schemas consist of pre-conditions (similar to the precondition-list of STRIPS) and effects (similar to a combination of add-list and delete-list of STRIPS) given by atoms $p(x_1, \dots, x_k)$ where p is a predicate symbol of arity k with arguments $x_1, \dots, x_k$. The instance describes the initial and goal situations, consisting of a set of objects O and the sets *Init* and *Goal* of grounded atoms, representing the initial and the goal situation, respectively.

A PDDL domain also defines the requirements, which are extensions to the language. It is important to specify these extensions because planners or other programs might not support them. Requirements appearing in this thesis are:

- **`:strips`** allows the usage of STRIPS-like effects. It is the most basic requirement and always assumed to be included.
- **`:typing`** allows specifying types for constants, predicates, and action parameters.
- **`:universal-preconditions`** allows preconditions to be quantified with `forall`.
- **`:conditional-effects`** allows effects to be applied conditionally with `when`.
- **`:negative-preconditions`** allows preconditions to be negated with `not`.

2.1.2 Generalized Planning

Besides finding a concrete plan for a specific instance, it is also possible to obtain a strategy, also called a policy, capable of solving many or even all instances in a given domain. The strategy must work reactively, meaning it cannot involve searching or planning. A policy that solves all (solvable) instances of a domain is called a general policy; if it solves them optimally, it is called an optimal policy. This field is called generalized planning [9] [28]. One crucial aspect of generalized planning is how policies are represented. They cannot be mappings from states to ground actions because the ground actions change based on the instance and its objects [28]. The way how policies are represented in our case will be introduced later. It is also important to mention that not all domains have a general or optimal policy. A typical trade-off in generalized planning is the trade-off between optimality and generalization. Often, when trying to increase the generalization of a policy, the optimality decreases, and vice versa, which is also present in the work of Ståhlberg *et al.* [28] and will be discussed later.

Classical planning offers meaningful language for representing instances and domains. In many cases, proving that an obtained policy is correct, general, or even optimal is possible. However, these approaches have limitations. There is heavy reliance on human-provided encodings of domains and instances mostly done in PDDL, requiring complete knowledge of the environment, which is not always available. Additionally, classical planning often does not scale well with the size of the domain and the increasing number of states [28]. The scalability problem means the training instances must be small enough to be solvable in a reasonable time frame.

2.2 Reinforcement Learning

Another approach in AI for obtaining policies is reinforcement learning (RL) [30]. In RL, an agent interacts with an environment by taking actions and receiving rewards for them. The goal is to learn a policy that maximizes the expected cumulative reward, which happens by trial and error. The following uses the notation and definitions of Sutton and Barto [30] with slight modifications.

2.2.1 Markov Decision Processes

An RL problem is described as Markov decision process (MDP) [30] which contains:

- $\mathcal{S}$ the set of states,
- $\mathcal{A}$ the set of actions,
- $\mathcal{R} \subseteq \mathbb{R}$ the set of rewards,
- $r : \mathcal{S} \times \mathcal{A} \to \mathcal{R}$ the reward function, and
- $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ the transition probability function.

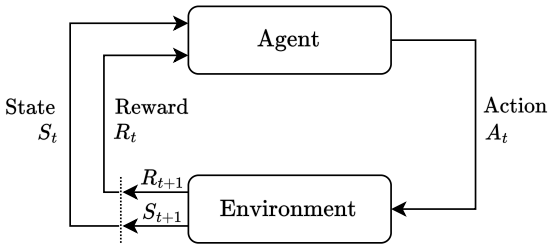


Figure 2.1: Agent-environment interaction in an MDP, adapted from Sutton and Barto [30]

The agent interacts with the environment in discrete time steps, as shown in Figure 2.1. At each time step $t = 0, 1, 2, \dots$, the agent observes the current state $S_t \in \mathcal{S}$ and chooses an action $A_t \in \mathcal{A}(s)$, where $\mathcal{A}(s)$ is the set of actions that are applicable in state s. One step later, the agent receives a reward $R_{t+1} \in \mathcal{R}$ and a new state $S_{t+1} \in S$. This process generates a trajectory: $S_0, A_0, R_0, S_1, A_1, R_1, S_2, A_2, R_2, \dots$. If the sets $\mathcal{S}, \mathcal{A}$ and $\mathcal{R}$ are finite the MDP is also called finite [30]. In this work, we consider only finite MDPs. The transition probability function p gives the probability of transitioning to state s' and getting reward r when taking action a in state s:

$$p(s', r \mid s, a) \doteq \mathbb{P}[S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a] \tag{2.1}$$

One crucial assumption of MDPs is the Markov property [30]. It states that the future is independent of the past, given the present. This means that states hold all relevant information about the past agent-environment interaction that will make a difference in the future. The Markov property is the base of RL algorithms and is formulated as:

$$\mathbb{P}[S_{t+1} \mid S_t, A_t] = \mathbb{P}[S_{t+1} \mid S_t, A_t, S_{t-1}, A_{t-1}, \dots] \tag{2.2}$$

A common trade-off in RL is between exploration vs. exploitation [30]. The agent must explore the environment to learn about it, but it must also exploit its current knowledge to maximize the expected cumulative reward. Balancing these two is essential for RL algorithms.

2.2.2 Returns, Episodes, and Discounting

The agent-environment interaction is often dividable into subsequences called episodes [30]. An episode is a finite sequence of states ending in a terminal state. The terminal state completes the current episode, and an independent initial state for the next episode follows. The length of an episode is T. In RL, the goal at each time step t is to maximize the expected cumulative reward G_t. The cumulative reward is the sum of all rewards from time step t to the end of the episode:

$$G_t \doteq R_{t+1} + R_{t+2} + \dots + R_T \tag{2.3}$$

However, as in many other cases, the agent-environment interaction is infinite and cannot be divided into episodes [30]. Then, the agent-environment interaction is continuing, and $T = \infty$, which is problematic because the sum of Equation (2.3) can be infinite. To solve this problem, a factor $\gamma \in [0, 1]$ discounts G_t and makes the sum of the rewards finite:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.4}$$

The discount factor γ balances the importance of immediate and future rewards. As γ approaches zero, the agent cares more and more about the immediate rewards, and if $\gamma = 0$, only the immediate reward matters. Conversely, as γ approaches one, the agent cares more and more equally about future rewards, and if $\gamma = 1$, the agent values all future rewards equally.

2.2.3 Policies, Value Functions, and Bellman Equations

A policy [30] is a mapping from states to probabilities of selecting each possible action. In a stochastic policy π, the probability of taking action a in state s is $\pi(a \mid s) \in [0, 1]$. A policy can also be deterministic, where $\pi(s)$ represents the action taken in state s. The goal of RL is to find an optimal policy π_* which maximizes the expected cumulative reward. There may be multiple optimal policies.

Another related fundamental concept are value functions [30]. The state-value function $v_\pi(s)$ of a policy π is the expected return when starting in state s and following policy π:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right] \tag{2.5}$$

where $\mathbb{E}_\pi[\cdot]$ is the expected value of a random variable under policy π. Similarly, the action-value function $q_\pi(s, a)$ is the expected return when starting in state s, taking action a and then following policy π:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right] \tag{2.6}$$

One important observation is that the expected cumulative reward G_t in Equation (2.4) can be expressed recursively as: $G_t = R_{t+1} + \gamma G_{t+1}$. Thus, the state-value functions from Equation (2.5) can also be expressed recursively:

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_{a \in \mathcal{A}(s)} \pi(a \mid s) \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r \mid s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']] \\
&= \sum_{a \in \mathcal{A}(s)} \pi(a \mid s) \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r \mid s, a)[r + \gamma v_\pi(s')]
\end{aligned}
\tag{2.7}
$$

This equation is known as the state-value Bellman equation [30]. It states that a state's value equals the discounted sum of its successor states' values and their rewards, weighted by the transition probabilities. Similarly, the action-value Bellman equation can be derived from Equation (2.6):

$$
\begin{aligned}
q_\pi(s, a) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\
&= \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r \mid s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']] \\
&= \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r \mid s, a)\left[r + \gamma \sum_{a' \in \mathcal{A}(s')} \pi(a' \mid s') \mathbb{E}_\pi[G_{t+1}] \,\middle|\, S_{t+1} = s', A_{t+1} = a'\right] \\
&= \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r \mid s, a)\left[r + \gamma \sum_{a' \in \mathcal{A}(s')} \pi(a' \mid s') q_\pi(s', a')\right]
\end{aligned}
\tag{2.8}
$$

All optimal policies share the same state-value and action-value function, called the optimal state-value function v_* and the optimal action-value function q_*, respectively. They are defined as:

$$
v_*(s) \doteq \max_\pi v_\pi(s) \tag{2.9}
$$

$$
q_*(s, a) \doteq \max_\pi q_\pi(s, a) \tag{2.10}
$$

This allows us to express the Bellman equations for the optimal value functions without referencing a policy. The optimal state-value Bellman equation [30] is:

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_{a \in \mathcal{A}} \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_{a \in \mathcal{A}} \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\
&= \max_{a \in \mathcal{A}} \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_{a \in \mathcal{A}} \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r \mid s, a)[r + \gamma v_*(s')]
\end{aligned}
\tag{2.11}
$$

Similarly, the optimal action-value Bellman equation [30] is:

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\
&= \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r \mid s, a)[r + \gamma v_*(s')]
\end{aligned}
\tag{2.12}
$$

2.2.4 Algorithm Categories

RL algorithms can be categorized into the following dimensions [30].

Model-Based vs. Model-Free

Model-based methods include learning or getting a model of the environment. Specifically, learning a model involves estimating the transition probability function p and the reward function r. The model allows the agent to plan and simulate the environment without directly interacting with it. In contrast, model-free algorithms directly learn a policy or a value function without using a model.

On-Policy vs. Off-Policy

On-policy methods learn about the target policy while constantly following it. In contrast, off-policy methods additionally have a behavior policy followed during training. The behavior policy is used to learn a different target policy.

Value-Based vs. Policy-Based vs. Actor-Critic

Value-based methods focus on estimating the value function, which is then used to derive a policy. Policy-based algorithms directly optimize a parameterized policy. Although the policy

in policy-based methods selects actions without consulting a value function, the method still may utilize a value function to optimize the policy's parameters.

Actor-critic methods combine the advantages of value-based and policy-based methods. Thus, they learn an approximation for the policy and the value function. The actor refers to the policy, and the critic refers to the value function.

2.2.5 Dynamic Programming

Dynamic programming (DP) [30] is a class of algorithms that compute optimal policies given a perfect model of a finite MDP. Although DP is often impractical due to its assumption of having a perfect model and its high computational complexity, it remains significant for providing a theoretical foundation for many RL algorithms, including those relevant to this thesis. It offers two main algorithms for solving MDPs: policy iteration and value iteration.

Policy Iteration

Policy iteration (PI) [30] in RL is an iterative algorithm that computes an optimal policy by alternating between policy evaluation and policy improvement. Policy evaluation determines the value function v_π of the current policy π. Initially, the value function v^0 is set to zero for terminal states and arbitrary values for other states. It is then updated iteratively using the Bellman equation from Equation (2.7) as an update rule:

$$v^{k+1}(s) \doteq \sum_{a \in \mathcal{A}(s)} \pi(a \mid s) \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r \mid s, a)\big[r + \gamma v^k(s')\big] \tag{2.13}$$

This process always converges to v_π for $k \to \infty$ [30]. In the policy improvement step, a new greedy policy π' is computed, which selects the optimal action according to the current value function v_π:

$$\pi'(s) \doteq \arg\max_{a \in \mathcal{A}(s)} q_\pi(s, a) = \arg\max_{a \in \mathcal{A}(s)} \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r \mid s, a)[r + \gamma v_\pi(s')] \tag{2.14}$$

The improvement step is guaranteed to improve the policy or keep it unchanged if it is already optimal [30]. PI iterates between policy evaluation and improvement until the optimal policy is found. Since the number of policies is finite in finite MDPs, convergence is guaranteed in a finite number of steps [30].

Value Iteration

Value iteration (VI) [30] is based on the idea that PI still converges to the optimal policy even when policy evaluation is limited to only one update step. This leads to the VI algorithm, which combines the truncated policy evaluation and policy improvement into a single step:

$$v^{k+1}(s) \doteq \max_{a \in \mathcal{A}(s)} \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r \mid s, a)\big[r + \gamma v^k(s')\big] \tag{2.15}$$

This equation is the Bellman optimality equation from Equation (2.11) used as an update rule.

2.2.6 Approximating Value Functions

As already mentioned, DP does not scale well with the size of the domain and the increasing number of states. To address this, RL employs two dimensions of approximation methods: stochastic updates and function approximators. Following Ståhlberg *et al.* [28], this section and the next introduce standard approximation methods for both dimensions used by their algorithms. The approximated value function is $\hat{v}(s)$.

Approximation methods use stochastic samples instead of considering all states, actions, and corresponding successor states in the update rules [28]. This approach is essential when there are too many states or actions to consider, or when the transition probabilities and rewards are unknown. The sampled update rule updates the value function with a sampled state S and a corresponding sampled action A. The action is sampled from the current policy π with the probability $\pi(A \mid S)$. The successor state S' and the reward R are both provided by the environment and thus sampled with the probability of $p(S', R \mid S, A)$. Given these samples, $R + \gamma\hat{v}(S')$ is an unbiased estimate of $v_\pi(S)$ assuming infinite updates [28]. Unbiased means that the estimate's expected value is equal to the actual value: $\mathbb{E}_\pi[R + \gamma\hat{v}(S')] = v_\pi(S)$. This leads to the following update rule:

$$\hat{v}^{k+1}(S) \doteq R + \gamma\hat{v}^k(S') \tag{2.16}$$

Rather than directly updating the value function with the new estimate, the update rule can be modified to use a step size $\alpha \in \mathbb{R}^+$ to control the extent of the update. This step size or learning rate is a characteristic of stochastic approximation methods [28]. Equation (2.16) is a special case with $\alpha = 1$ of the following update rule:

$$\hat{v}^{k+1}(S) \doteq \hat{v}^k(S) + \alpha\big[R + \gamma\hat{v}^k(S') - \hat{v}^k(S)\big] \tag{2.17}$$

DP uses a table to store value functions [30], with each state having a dedicated value. As the number of states grows, this table becomes too large and infeasible to store and compute. Instead, it is possible to use a function approximator with adjustable parameters $\boldsymbol{\omega}$ to estimate the value function [28, 30]. Rather than updating a single table entry, the entire function's parameters are updated simultaneously. Additionally, the function approximator helps to generalize the value function to unseen states. The goal of an update step is to minimize the difference between our current estimate $\hat{v}(S, \boldsymbol{\omega})$ and the new estimate $R + \gamma\hat{v}(S', \boldsymbol{\omega})$ [28]. To achieve this, the squared loss function can be used:

$$\mathcal{L}(\boldsymbol{\omega}) \doteq \frac{1}{2}[R + \gamma\hat{v}(S', \boldsymbol{\omega}) - \hat{v}(S, \boldsymbol{\omega})]^2 \tag{2.18}$$

The update rule for the parameters $\boldsymbol{\omega}$ is then constructed with a step of gradient descent:

$$\begin{aligned}
\boldsymbol{\omega}^{k+1} &\doteq \boldsymbol{\omega}^k - \alpha\nabla L\big(\boldsymbol{\omega}^k\big) \\
&= \boldsymbol{\omega}^k - \alpha\nabla\Big[\frac{1}{2}\big[R + \gamma\hat{v}\big(S', \boldsymbol{\omega}^k\big) - \hat{v}\big(S, \boldsymbol{\omega}^k\big)\big]^2\Big] \\
&= \boldsymbol{\omega}^k - \alpha\big[R + \gamma\hat{v}\big(S', \boldsymbol{\omega}^k\big) - \hat{v}\big(S, \boldsymbol{\omega}^k\big)\big] \cdot \nabla\big[R + \gamma\hat{v}\big(S', \boldsymbol{\omega}^k\big) - \hat{v}\big(S, \boldsymbol{\omega}^k\big)\big] \\
&= \boldsymbol{\omega}^k + \alpha\big[R + \gamma\hat{v}\big(S', \boldsymbol{\omega}^k\big) - \hat{v}\big(S, \boldsymbol{\omega}^k\big)\big] \cdot \nabla\hat{v}\big(S, \boldsymbol{\omega}^k\big)
\end{aligned} \tag{2.19}$$

2.2.7 Policy Gradient Methods

Policy gradient methods [28, 30] are a subclass of policy-based RL algorithms that update the policies' adjustable parameters $\boldsymbol{\theta}$ via gradient descent steps. The performance of a parametrized policy $\pi_{\boldsymbol{\theta}}$ is defined as the expected cumulative reward across all start states $s_0 \in S_0$ weighted by the probability of starting them $h(s_0)$ [28, 30]:

$$J(\boldsymbol{\theta}) \doteq \sum_{s_0 \in \mathcal{S}_0} h(s_0)\mathbb{E}_{\pi_{\boldsymbol{\theta}}}[G_t \mid S_t = s_0] = \sum_{s_0 \in \mathcal{S}_0} h(s_0)v_{\pi_{\boldsymbol{\theta}}}(s_0) \tag{2.20}$$

Let $\mathbb{P}[s \to s', k, \pi_{\boldsymbol{\theta}}]$ be the probability of transitioning from state s to state s' in exactly k steps under policy $\pi_{\boldsymbol{\theta}}$. Defining $\eta_{\pi_{\boldsymbol{\theta}}}(s) = \sum_{s_0 \in \mathcal{S}_0} \sum_{k=0}^{\infty} \mathbb{P}[s_0 \to s, k, \pi_{\boldsymbol{\theta}}]$ allows us to express the probability of visiting state s under policy $\pi_{\boldsymbol{\theta}}$ as $\mu_{\pi_{\boldsymbol{\theta}}}(s) = \frac{\eta_{\pi_{\boldsymbol{\theta}}}(s)}{\sum_{s' \in \mathcal{S}} \eta_{\pi_{\boldsymbol{\theta}}}(s')}$. According to Sutton and Barto [30, Policy Gradient Theorem] the following holds:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_{s \in \mathcal{S}} \mu_{\pi_{\boldsymbol{\theta}}}(s) \sum_{a \in \mathcal{A}(s)} q_{\pi_{\boldsymbol{\theta}}}(s, a)\nabla\pi_{\boldsymbol{\theta}}(a \mid s) \tag{2.21}$$

This theorem can be extended with an arbitrary baseline $b(s)$ that is independent of a [30]:

$$\nabla J(\theta) \propto \sum_{s \in \mathcal{S}} \mu_{\pi_\theta}(s) \sum_{a \in \mathcal{A}(s)} \big[q_{\pi_\theta}(s,a) - b(s)\big] \nabla \pi_\theta(a \mid s) \tag{2.22}$$

The baseline reduces the variance of estimations while keeping the expectation the same. A common choice is the approximated value function $\hat{v}_{\pi_\theta}$ [28].

Similarly to the value functions in Section 2.2.6, this expression can be approximated by taking samples [28]. For example, $\big[q_{\pi_\theta}(S,A) - b(S)\big] \nabla \ln \pi_\theta(A \mid S)$ is an unbiased estimator of the right-hand side of Equation (2.22) assuming S is sampled with $\mu(S)$ and A is sampled with $\pi_\theta(A \mid S)$. Moreover, $q_{\pi_\theta}(S,A)$ can further be approximated with another unbiased estimator $R + v_{\pi_\theta}(S')$ where S' and R are sampled from the environment with $p(S', R \mid S, A)$. Thus the approximated value function $\hat{v}_{\pi_\theta}$ from Section 2.2.6 can be used to estimate the policy's gradient:

$$\widehat{\nabla J(\theta)} \doteq \big[R + v_{\pi_\theta}(S') - \hat{v}_{\pi_\theta}(S)\big] \nabla \ln \pi_\theta(A \mid S) \tag{2.23}$$

The update rule for the parameters θ is then constructed via a step of gradient ascent:

$$\begin{aligned} \theta^{k+1} &\doteq \theta^k + \alpha \widehat{\nabla J(\theta^k)} \\ &= \theta^k + \alpha \Big[R + v_{\pi_{\theta^k}}(S') - \hat{v}_{\pi_{\theta^k}}(S)\Big] \nabla \ln \pi_{\theta^k}(A \mid S) \end{aligned} \tag{2.24}$$

2.2.8 Advantages and Limitations

In contrast to classical planning, RL does not require symbolic states. For example, it is possible to represent the states as images. Additionally, there is no need for human-provided encodings of domains and instances. As shown in the agent-environment interaction of Figure 2.1, the environment is treated as a black box. Only the definitions for states, actions, and rewards are needed. This makes RL more versatile and applicable to a broader range of problems. However, RL lacks the benefit of having a meaningful formal language. Also, it is nearly impossible to prove that the obtained policy is correct, general, or optimal [28].

2.3 Artificial Neural Networks

A parametrized policy function in RL often relies on artificial neural networks (ANNs). In general, when an ANN is used in RL, we call it deep reinforcement learning (DRL) [30]. This section introduces three types of ANNs namely multilayer perceptrons in Section 2.3.1, convolutional neural networks in Section 2.3.1, and (relational) graph neural networks in Section 2.3.3.

2.3.1 Multilayer Perceptrons

One of the most basic types of ANNs are multilayer perceptrons (MLPs) [10]. An MLP consists of multiple perceptron layers arranged in a feedforward manner, meaning that the information only flows in one direction from the input layer to the output layer. A perceptron is a simple function that maps an input vector $\mathbf{x} \in \mathbb{R}^D$ to an output scalar $y(\mathbf{x}) \in \mathbb{R}$ using weights $\mathbf{w} \in \mathbb{R}^D$, a bias $w_0 \in \mathbb{R}$ and an arbitrary activation function $g : \mathbb{R} \to \mathbb{R}$ where the weights and the bias are parameters of the perceptron:

$$y(\mathbf{x}) = g\big(\mathbf{w}^T\mathbf{x} + w_0\big) = g\left(\sum_{i=1}^{D} w_i x_i + w_0\right) \tag{2.25}$$

This single perceptron, which outputs one class, can be extended to multiple classes K. The output is then defined as $y(\mathbf{x}) = \langle y_1(\mathbf{x}), \dots, y_K(\mathbf{x})\rangle \in \mathbb{R}^K$ where for each class $k \in \{1, \dots, K\}$:

$$y_k(x) = g\big(\mathbf{w}_k^T\mathbf{x} + w_{k0}\big) = g\left(\sum_{i=1}^{D} w_{ki} x_i + w_{k0}\right) \tag{2.26}$$

One limitation of multi-class perceptrons is their inability to model non-linear functions [10]. This can be addressed by stacking multiple perceptrons $y_1, \dots, y_L$ into at least two layers L and using non-linear activation functions between the layers to create the so-called MLP. Its output is calculated by sequentially applying the layers:

$$\mathrm{MLP}(\mathbf{x}) = y_n(\dots(y_2(y_1(\mathbf{x})))) \tag{2.27}$$

2.3.2 Convolutional Neural Networks

Another type of feedforward ANN are convolutional neural networks (CNNs) [10]. CNNs process structured grid-like data and are well-suited for processing images represented as a two-dimensional grid of pixels. They are particularly good at recognizing patterns, shapes, and structures within images.

In contrast to MLPs, CNNs retain and exploit the spatial structure of the input data [10]. This is achieved by using convolutional layers, which act as filters applied to small regions that slide over the input data. Pooling layers then summarize the information and reduce the dimensionality of the convolutional layer. After possibly multiple convolutional and pooling layers, the final output is flattened and passed to a traditional MLP. One downside of CNNs is that the architecture fixes the size of the input images.

2.3.3 Graph Neural Networks

A further type of ANNs are graph neural networks (GNNs) [26], which process graph-structured data. Unlike MLPs and CNNs, which handle fixed-size tabular data like vectors, images, or sequences, GNNs operate on graphs of varying sizes with complex relationships. A key property of GNNs is their invariance to graph isomorphism. They are applied to tasks like node classification, link prediction, graph classification, and more.

A graph is defined by $G = \langle V, E \rangle$ where V is the set of vertices and $E \in V \times V$ is the set of edges of the graph. In our case, the graph is undirected and unweighted. To accomplish its functionality, a GNN utilizes a message passing scheme [26, 28, 29]. For this, a feature vector $f_i(v) \in \mathbb{R}^k$ is assigned to each node $v \in V$ where $i \in \{0, \dots, L\}$, and L is the number of layers. k and L are hyperparameters of the GNN. In our case, all feature vectors are initialized as all zeros in the beginning. Sequentially, for $i = 1, \dots, L$, the GNN updates the feature vectors of the nodes by passing messages between the nodes. The message-passing scheme is defined as:

$$f_{i+1}(v) \doteq \mathrm{comb}_i\Big(f_i(v), \mathrm{agg}_i\{f_i(v') \mid v' \in N(v)\}\Big) \tag{2.28}$$

where $N(v)$ is the set of neighbors of v. The aggregation functions agg_i (for example, sum or max) condenses arbitrary many vectors into a single vector. The combination functions comb_i merge pairs of vectors. Both types are parametrized learnable functions. A single step of message passing can be seen as one node receiving messages from its neighbors, aggregating them, and then combining them with its feature vector, thus updating its feature vector. In the end, the final feature vectors $f_L(v)$ for every node $v \in V$ are left. The final output of the GNN is then calculated by a learnable function F called the read-out function, which takes the final feature vectors of the nodes as input: $F(\{f_L(v) \mid v \in V\})$.

Relational Graph Neural Networks

One variant of GNNs are relational graph neural networks (R-GNNs) [28, 29]. Instead of regular graphs, R-GNNs operate on relational structures. A relational structure is defined as $\langle A, \{R_1, \dots, R_k\} \rangle$ where A is a non-empty set, the universe, and $\{R_1, \dots, R_k\}$ are relations on A. Each relation R is a subset of $A^{\mathrm{arity}(R)}$ where $\mathrm{arity}(R) \in \mathbb{N}$ is the arity of R. Standard graphs $G = \langle V, E \rangle$ are a special case of relational structures where the universe is the set of vertices V, and the only relation are the edges E, with arity two.

To modify GNNs for relational structures, the message passing scheme from Equation (2.28) must be adapted [28]. Each relation R of the structure defines hyperedges that simultaneously connect $\mathrm{arity}(R)$ many objects in the universe. Thus, instead of passing messages between two neighboring vertices, messages are passed between all objects connected by a relation. As a result, each object $o \in A$ receives messages

$$m_{q,o} \doteq \left[\mathrm{comb}_R\big(f_i(o_1), \dots, f_i(o_{\mathrm{arity}(R)})\big)\right]_j \tag{2.29}$$

from each hyperedge $q \doteq \langle o_1, \dots, o_{\mathrm{arity}(R)} \rangle \in R$ it is connected to, meaning $o \in o_1, \dots, o_{\mathrm{arity}(R)}$. The combination function comb_R is specific to the relation R and $[\cdot]_j$ denotes the j-th element of the vector inside it where j is the index of the object o in the hyperedge q. Once per layer, the messages are aggregated and combined with the current feature vector:

$$f_{i+1}(o) \doteq \mathrm{comb}_i\Big(f_i(o), \mathrm{agg}_i\big(\{m_{q,o} \mid q \in R, R \in \{R_1, \dots, R_k\} \text{ if } o \in q\}\big)\Big) \tag{2.30}$$

Expressivity of Graph Neural Networks

One drawback of regular GNNs and R-GNNs is their limited expressivity. Specifically, their expressivity lies between GC_2 and C_2 [2, 12, 16, 28], meaning that they can express anything within GC_2 but not everything in C_2, thus: $\mathsf{GC}_2 \subset \mathrm{GNN} \subset \mathsf{C}_2$

The counting logic C introduces counting quantifiers $\exists^{\geq p} x\, \varphi(x)$ to FO, which require at least p different objects to satisfy φ. It has the same expressive power as FO since the counting quantifier can be converted to FO as follows [12]:

$$\exists^{\geq p} x\, \varphi(x) \equiv \exists x_1 \dots \exists x_p \left(\bigwedge_{1 \leq i < j \leq p} x_i \neq x_j \wedge \bigwedge_{i=1}^{p} \varphi(x_i) \right) \tag{2.31}$$

However, this conversion increases the required variables depending on p [12]. Thus, when FO and C are restricted to use at most k variables, denoted as FO_k or C_k, they differ in expressive power. For instance, a formula indicating that there are at least three different objects in the universe can be easily expressed in C_2 (by $\exists^{\geq 3} x\, (x = x)$), but not in FO_2, which requires at least three variables.

The guarded fragment GC of C restricts all quantifiers to be of the form $\exists^{\geq p} y(E(x, y) \wedge \psi(x))$ where x and y are distinct variables, E is a binary relation, and x appears as free-variable in the formula ψ [12]. The standard existential and universal quantifiers are a special case of the counting quantifiers and can be expresses as $\exists x\, \varphi(x) \equiv \exists^{\geq 1} x\, \varphi(x)$ and $\forall x\, \varphi(x) \equiv \neg \exists^{\geq 1} x\, \neg \varphi(x)$ respectively. Thus the universal quantification in GC is always logically equivalent to the form $\forall y(E(x, y) \rightarrow \psi'(x))$, as shown by the following equivalence:

$$\underbrace{\neg \exists^{\geq 1} y(E(x, y) \wedge \psi(x))}_{\mathsf{GC}\text{ formula}} \equiv \forall y(\neg E(x, y) \vee \neg \psi(x)) \equiv \forall y(E(x, y) \rightarrow \psi'(x)) \tag{2.32}$$

where $\psi(x) \equiv \neg \psi'(x)$, and the other symbols defined as above. GC_2 is the fragment of GC where the formulas are restricted to use at most two variables.

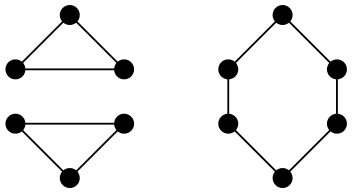


Figure 2.2: In C_2 indistinguishable graphs, taken from Horčík and Šír [16]

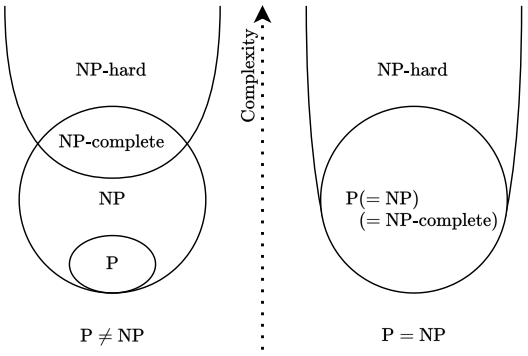


Figure 2.3: Euler diagram of the P vs. NP problem, adapted from Esfahbod [6]

2.4 P vs. NP

The complexity classes P and NP [4] are fundamental concepts in computer science. P is the class of decision problems that can be solved by a deterministic algorithm in polynomial time. NP is the class of decision problems that can be solved by a non-deterministic algorithm in polynomial time. Whether P equals NP is still a significant open question. A problem is said to be NP-hard if every problem in NP can be reduced to it in polynomial time. A problem is said to be NP-complete if it is both NP-hard and in NP.

3 Related Work

This chapter reviews the related work. First, we examine the work and algorithms of Ståhlberg *et al.* [28]. Next, we introduce PushWorld [17], the domain of our focus, along with its challenges and the tried approaches in classical planning and reinforcement learning. Finally, we describe the Sokoban domain and its similarities to PushWorld, which enable us to apply a different version of its PDDL definitions instead of PushWorld's more complex ones.

3.1 Learning General Policies with Policy Gradient Methods

The work of Ståhlberg *et al.* [28] combines classical planning and reinforcement learning. An RL agent is trained on symbolic states provided by a classical planning model encoded in PDDL. The instances, domains, and states are thus represented by meaningful language. Additionally, since RL is used for obtaining the policy, there is a reduced scalability problem compared to combinatorial methods in classical planning.

3.1.1 The Two Actor-Critic Algorithms

Ståhlberg *et al.* [28] employ two primary algorithms for learning: a standard actor-critic algorithm (Algorithm 1) and an all-actions actor-critic algorithm (Algorithm 2). As mentioned, actor-critic means these methods learn an approximation for the policy and the value function. The policy is parametrized by θ and the value function by ω. Both have their corresponding step size α and β. The algorithms are centered around an endless loop that indefinitely updates θ and ω.

The foundation of both algorithms has already been laid in Section 2.2.6 and Section 2.2.7. Algorithm 1 simply combines the update rules from Equation (2.19) and Equation (2.24). In addition to the sampling strategies discussed in the just mentioned sections, Algorithm 1 also samples an MDP because the training is done on possibly multiple MDPs. At the end of each loop iteration, it is ensured that the values of goal states remain at zero.

Algorithm 2 is very similar to Algorithm 1 but with one key difference. Instead of sampling a successor state, it sums over all successor states weighted by the probability of transitioning to them. It also incorporates a baseline. The additional information in each update step is supposed to provide better estimates and more frequent updates than Algorithm 1 [28]. However, to compute the sum over all successor states, one has to know them, which is achieved by consulting the underlying model. This makes Algorithm 2 model-based, while

Algorithm 1 is model-free as it does not need to access all the successor states. Note that the softmax function from Equation (2.24) also needs to know all the possible successor states, which technically also makes Algorithm 1 model-based.

In both algorithms, a fixed reward of -1 is assumed. This means that the sole goal is to minimize the number of steps taken to reach a goal state. This also addresses the optimality/generalization trade-off by avoiding cycles, thus increasing the chance of reaching a goal state favoring generalization like Ståhlberg *et al.* [28] mention. They also increase generalization by taking uniform samples from the state space, meaning the policy is trained to work from all states, not just initial ones. Another difference to the provided background is that their policy $\pi(s' \mid s)$ maps states into other states instead of actions due to grounded actions varying based on objects in an MDP.

Algorithm 1 Standard Actor-Critic for generalized planning from Ståhlberg *et al.* [28]: Successor states s' sampled with probability $\pi(s' \mid s)$.

Input: Training MDPs $\{M_i\}_i$, each with state priors p_i
Differentiable policy $\pi(s \mid s')$ with parameter θ
Differentiable value function $V(s)$ with parameter ω
Step sizes $\alpha, \beta > 0$, discount factor λ

Initialize parameters θ and ω
loop
  Sample MDP index $i \in \{1, \dots, N\}$
  Sample non-goal state S in M_i with probability p_i
  Sample successor state S' with probability $\pi(S' \mid S)$
  Let $\delta \leftarrow 1 + \gamma V(S') - V(S)$
  $\omega \leftarrow \omega + \beta\delta\nabla V(S)$
  $\theta \leftarrow \theta - \alpha\delta\nabla \log \pi(S' \mid S)$
  If S' is a goal state, $\omega \leftarrow \omega - \beta V(S')\nabla V(S')$

Algorithm 2 All-Actions Actor-Critic from Ståhlberg *et al.* [28]: All successor states considered for updating policy and value functions.

Input: Training MDPs $\{M_i\}_i$, each with state priors p_i
Differentiable policy $\pi(s \mid s')$ with parameter θ
Differentiable value function $V(s)$ with parameter ω
Step sizes $\alpha, \beta > 0$, discount factor λ

Initialize parameters θ and ω
loop
  Sample MDP index $i \in \{1, \dots, N\}$
  Sample non-goal state S in M_i with probability p_i
  Let $V' \leftarrow 1 + \gamma \sum_{s' \in N(S)} [\pi(s' \mid S) V(s')]$
  Let $b(S) \leftarrow V' - 1$ be the baseline
  $\omega \leftarrow \omega + \beta(V' - V(S))\nabla V(S)$
  $\theta \leftarrow \theta - \alpha \sum_{s' \in N(S)} [(V(s') - b(S))\nabla \pi(s' \mid S)]$
  If $s' \in N(S)$ is a goal state, $\omega \leftarrow \omega - \beta V(s')\nabla V(s')$

3.1.2 Constructing the Value and Policy Function

Ståhlberg *et al.* [28] use a single R-GNN as the core component of the value and policy function. It takes a state as an input and outputs a single scalar value representing its goodness. To pass a state, a set of positive grounded predicates, into a R-GNN, it must be viewed as or transformed into a relational structure. Here, the universe is equivalent to all objects in the problem. Each predicate is equivalent to a relation containing all positive groundings. So, for example, if $p(o_1, \dots, o_n)$ is a positive grounded predicate, the relational structure contains a relation p with an element $\langle o_1, \dots, o_n \rangle \in p$ where $o_1, \dots, o_n$ are objects in the universe.

Since the same R-GNN serves as the core component of the value and policy function, they use the same object embeddings $f^s(o)$ for each object o in state s at the last layer with only the read-out functions differing [28]. The value function $V(s)$ is constructed by summing up all object embeddings and passing them through an MLP which outputs a single scalar:

$$V(s) \doteq \mathrm{MLP}\left(\sum_{o \in O} f^s(o)\right) \tag{3.1}$$

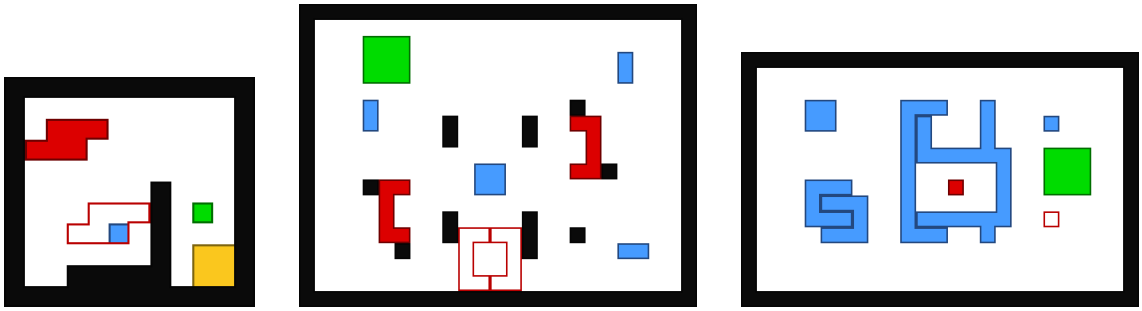
Remember that the policy maps a state to a successor state in their work. Therefore, all possible successors need to be evaluated. Based on this, the successor can be chosen deterministically by taking the one with the highest value or stochastically by calculating a discrete probability density from the values and selecting accordingly. The quality of a transition from s to s' is $\mathrm{logit}(s' \mid s) \in \mathbb{R}$ which is defined as:

$$\mathrm{logit}(s' \mid s) \doteq \mathrm{MLP}\left(\sum_{o \in O} \mathrm{MLP}\Big(f^s(o), f^{s'}(o)\Big)\right) \tag{3.2}$$

The outer MLP outputs a single scalar, while the inner one outputs a vector of size $2k$, where k is the dimension of a single object embedding. The purpose of the latter is to derive new features specific to transitions. A probability distribution can be determined based on the logits of all possible state transitions, which is done by the softmax function:

$$\pi(s' \mid s) \doteq \mathrm{softmax}(\mathrm{logit}(s' \mid s)) = \frac{\exp(\mathrm{logit}(s' \mid s))}{\sum_{s' \in N(s)} \exp(\mathrm{logit}(s' \mid s))} \in [0, 1] \tag{3.3}$$

where $N(s)$ is the set of all successor of state s. Note that the MLPs in Equation (3.1) and both in Equation (3.2) all have different parameters. Only the parameters of the R-GNN are shared.

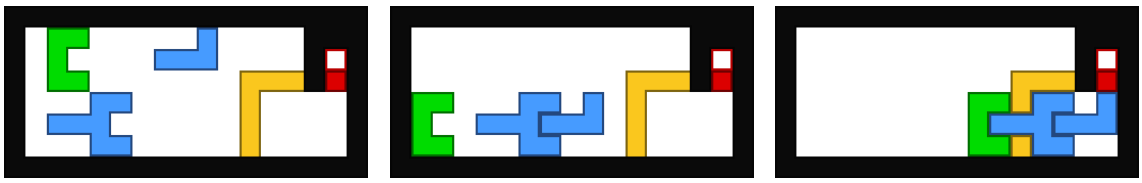


(a) Get Out Of My Spot (Level 1, 20 steps)

(b) Armor (Level 3, 86 steps)

(c) Captive (Level 4, 149 steps)

Figure 3.1: Three distinct PushWorld puzzles taken from Kansky *et al.* [17] with their puzzle name, level, and optimal plan length.



(a) Initial state

(b) Step 1: Assemble tool

(c) Step 2: Use tool

Figure 3.2: An example of a PushWorld puzzle that involves tool use. After step two, the puzzle can be solved with a single up action, which is not shown here.

3.2 PushWorld: A Benchmark for Manipulation Planning

PushWorld from Kansky *et al.* [17] is a two-dimensional physical environment with finite, discrete states. It is fully observable, deterministic, and static. Static means that the environment does not change while deciding on a new action.

PushWorld's environment consists of a grid of cells containing various rigid objects with various shapes and sizes. Each object is of type agent (green), box (blue or red), wall (black) or barrier (yellow), as shown in the example puzzles in Figure 3.1. A single agent object always moves one grid cell up, down, left, or right per time step. During its move, the agent can push boxes in front of it, with no limit to the number of boxes that can be pushed simultaneously. Objects move only if they the agent pushes them at the current time step, and they cannot rotate. Walls are immovable and impenetrable by the agent or any pushed objects. Barriers are also immovable and impenetrable by the agent, but boxes can be pushed through the barriers as if they were not there. The goal in PushWorld is to push all red boxes to their designated goal positions (marked by a red outline). In the following, we call the blue boxes normal boxes and the red boxes goal boxes. A puzzle is solved when all goals are met. Most puzzles contain dead-ends where the agent can no longer solve the puzzle.

3.2.1 Challenges of the Environment

The PushWorld environment is designed to present and combine several challenges. Among others, Kansky *et al.* [17] highlight the following:

- **Path Planning:** The agent must find a collision-free path to move itself and one or multiple boxes between positions. It must plan its path to avoid obstacles and dead-ends while also considering the different shapes and sizes of the involved objects.
- **Moveable Obstacles:** The agent must decide whether to move an obstacle out of the way or navigate around it. This decision can be irreversible, such as pushing a box into a corner. Additionally, boxes can behave parasitically, irreversibly sticking to each other and complicating the separation process like the assembled box in Figure 3.2b, which is not separable with the given agent shape.
- **Tool Use:** The agent may need to use boxes (tools) to move other boxes to their required positions. Tools sometimes must be assembled like in Figure 3.2. This also means that the role of a box might change during the puzzle. Depending on the current state, a box might be a tool, an obstacle, or a goal.
- **Prioritizing Multiple Goals:** Puzzles can require achieving multiple goals, often simultaneously or in a specific sequence. For example, arranging multiple boxes to push them simultaneously into their goals may be required. So, the agent must carefully plan the order of achieving these goals, as moving an object to its goal too early may block the path to other goals.

Despite the challenges of the environment itself, there are also challenges mentioned by Kansky *et al.* [17] specifically to RL algorithms:

- **Sparse Rewards:** Rewards are only provided upon reaching the goal. Because of the delayed feedback, in the beginning the agent needs to explore the environment extensively without any guidance.
- **Credit Assignment:** It also makes it hard to decide which actions were beneficial and contributed to reaching the goal and which were not.
- **Exploration:** The agent also must avoid dead-ends during exploration.
- **Hierarchy:** Humans tend to separate a puzzle into sub-problems (for example, obtaining tools or freeing a path), which agents might need to do as well.

These challenges make the environment engaging as an addition to the more common ones currently used by Ståhlberg *et al.* [28]. As we will see, it is more complex and provides excellent opportunities for testing the limited expressivity of the R-GNNs.

3.2.2 PushWorld PDDL

Kansky *et al.* [17] also provide PDDL definitions. The full PDDL domain can be found in Listing A.1. In the following objects, refer to the agent or boxes. A position is a cell in the grid, and a direction is either up, down, left, or right. The domain defines the following predicates:

- **`connected(?from, ?to, ?dir):`** Indicates that position `?to` is reachable from another position `?from` given direction `?dir`. This predicate is used to define the grid shape. It is static, which means that no action changes it.
- **`at(?obj, ?pos):`** Denotes the current position `?pos` of object `?obj`. Used to define each object's initial position and keep track of it. Although objects can occupy multiple cells, only one cell represents the object's position. This cell is chosen arbitrarily from the occupied cells. The following two predicates implicitly give the other cells.
- **`wall-collision(?obj, ?next-pos):`** Indicates that object `?obj` would be colliding with a wall or barrier if it was at position `?next-pos` excluding the implicit outer wall. So, the predicate declares invalid object positions. It is static and initialized by a preprocessor iterating over all possible object positions and checking their validity.
- **`in-collision(?obj, ?pos, ?other-obj, ?other-pos):`** Indicates that object `?obj` at position `?pos` would be colliding with another object `?other-obj` at position `?other-pos`. Similar to `wall-collision`, this predicate also indicates invalid positions of objects. It is also static and initialized by a preprocessor iterating over all possible object pairs and positions and checking if they are valid.
- **`should-move(?obj):`** Marks that object `?obj` should move within the current time step. This predicate is used as a helper for the action mechanism, which is explained in the next paragraph.
- **`has-moved(?obj):`** Marks that object `?obj` has already been moved within the current time step. Also a helper for the action mechanism, which is explained in the next paragraph.

The last two mentioned predicates are used for the action mechanism. Because in PushWorld, the agent can push multiple objects simultaneously, the domain cannot easily be defined by just one action for one move. Instead, a rather complicated definition of the action is needed, which consists of two counterparts:

- **`move-agent(?dir):`** This action initiates the movement of the agent in the given direction `?dir`. It can only be executed if no object is marked with `should-move`. When executed, it marks the agent with `should-move` in the given direction.
- **`push(?obj, ?dir, ?pos, ?next-pos):`** Moves an object `?obj` that is marked as `should-move` one cell in the given direction `?dir` from position `?pos` to `?next-pos`. The object gets marked as `has-moved`, and all connected objects that also should move as a consequence get marked as `should-move`.

Initially, the agent is marked with `should-move` by the `move-agent` action. The `push` action is then repeatedly executed to move one object at a time until none remain marked with `should-move`. The `has-moved` predicate ensures that an object is only moved once per time step. A drawback of this definition is the inability to know beforehand if a move is possible. We can only determine this after execution; if unsuccessful, no further action can be taken, leaving the agent stuck in an in-between state.

An unusual observation is that the domain defines constants $m1, m2, \ldots$ for each object. These constants are then directly used for the action definitions. Thus, different domain definitions are needed based on the amount of moveable objects in an instance. This means that the domain is not entirely independent of the actual instance, which should generally be the case as the domain should be general and reusable.

Finally, there are also PDDL instance definitions, where the instance defines the grid shape and the initial position of objects via the `connected` and `at` predicate. The object shape is implicitly given by precomputing all possible colliding positions between pairs of objects, walls, and barriers, which are then stored in the `wall-collision` and `in-collision` predicates.

3.2.3 State of the Art with Planning

The PDDL definitions of PushWorld are also used by Kansky *et al.* [17] to solve puzzles with classical planning. They introduce a new heuristic called recursive graph distance (RGD) [17], based on the context-enhanced additive heuristic (CEA) [14]. RGD combines strengths of other heuristics like constrained movement, efficient path planning, or tool manipulation. It can also be modified by combining it with the novelty heuristic [20] to form Novelty+RGD [17]. Both versions are used in a greedy best-first planner.

RGD is compared to a variety of different classical planners [17]: Fast Forward, Fast Downward, LAMA, Best-First Width Search, Fast Downward Stone Soup, and Saarplan. These planners, along RGD and Novelty+RGD, are tested against a handful of PushWorld benchmark puzzles by Kansky *et al.* [17], divided into different categories based on their difficulty:

- **Level 1** puzzles can be solved by a human in a few seconds,
- **Level 2** puzzles in a few minutes,
- **Level 3** puzzles in 15 minutes and
- **Level 4** puzzles in more than 15 minutes.

The planners are tested simultaneously against all levels. The main metric is the number of puzzles solved against the maximal solve time per puzzle. Novelty+RGD outperforms all other planners with a maximum solving time of 1000 seconds, solving about 70% of all puzzles [17].

3.2.4 State of the Art with Reinforcement Learning

Besides the classical planning approaches of Kansky *et al.* [17], there are attempts to solve the puzzles with RL algorithms. PushWorld was implemented as an OpenAI Gym [3] environment and as a Google DeepMind RL [23] environment. Two distinct RL algorithms were used to solve the puzzles: Deep Q-Network (DQN) [22] and Proximal Policy Optimization (PPO) [27]. DQN is a model-free, off-policy algorithm for value-based RL, while PPO is a model-free, on-policy, policy-gradient algorithm. Both used the implementation of the Google DeepMind research framework for RL named Acme [15] with a CNN as the function approximator. The observation space is an RGB image represented as a tensor $\langle \text{height}, \text{width}, 3 \rangle$, where the last dimension portrays the three RGB channels. The action space is a discrete space of size four representing the directions up, down, left, and right. The agent receives a reward of $+1$ for moving an object onto its goal position and -1 for moving an object off its goal position. When all goals are satisfied, the agent receives a reward of $+10$. Each action costs 0.01.

The Level 1 to 4 benchmark puzzles are too complex and too few to be used for training RL algorithms. Thus, the authors also provide simpler, procedural-generated puzzles, named Level 0 and divided into train and test sets. Level 0 puzzles are further grouped into different categories: BASE, LARGERPUZZLESIZES, MOREWALLS, MOREOBSTACLES, MORESHAPES, MULTIPLEGOALS and ALL. The BASE category contains puzzles of size 5×5 with only one goal, and objects have only sizes 1×1. The other categories contain puzzles that introduce additional challenges compared to BASE hinted at by their names. LARGERPUZZLESIZES requires longer plans to solve the puzzles. MOREWALLS and MOREOBSTACLES make the instance more complex by adding more objects to interact with. MORESHAPES introduces an entirely new aspect of the environment by having objects of different shapes than 1×1. In MULTIPLEGOALS, the agent potentially has to prioritize goals, and the plans get more complex and increase in size. The ALL category combines all changes from the other categories.

Both DQN and PPO were trained for 450 million time steps with Level 0 puzzles. The authors Kansky *et al.* [17] highlight that the training accuracy of PPO is significantly higher than the test accuracy, suggesting overfitting. DQN shows less difference between training and test accuracy. DQN and PPO show the best performance on LARGERPUZZLESIZES and the worst performance on MULTIPLEGOALS, likely due to the increased difficulty and importance of goal prioritization. They also noticed a low performance of ALL and suspected the high diversity of puzzles might be the reason. They also attempted to train on Level 1 puzzles where DQN only managed to solve less than 1% of the puzzles and PPO about 6%. Due to this low performance, higher levels were not attempted.

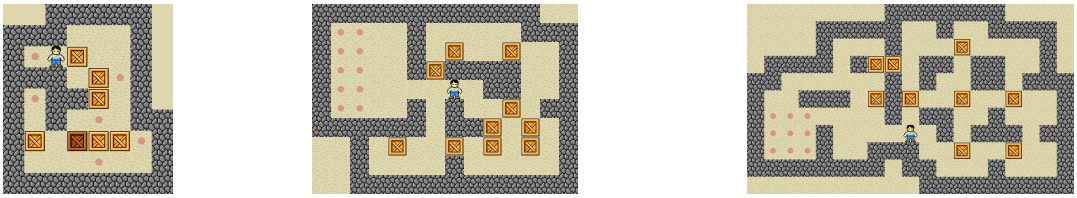


Figure 3.3: Three examples of Sokoban puzzles (from `mathsisfun.com/games/sokoban`)

3.3 Sokoban

Sokoban (Japanese for warehouse keeper) is a well-known puzzle game. Due to its complexity, it is often used as a benchmark environment. Like PushWorld, Sokoban is a two-dimensional grid-based environment. The character ( ) also moves up, down, left, and right. The goal is to push all boxes ( ) onto goal positions ( ). The player can only push one box at a time and cannot pull boxes. Neither the player nor the boxes can move through walls ( ). Example puzzles can be seen in Figure 3.3. The game is fully observable, deterministic, and static. Sokoban is known to be NP-hard [5].

Despite the similarities between Sokoban and PushWorld, there are notable differences:

- Sokoban does not have normal boxes (boxes without a goal).
- Sokoban does not have barriers.
- In Sokoban, all boxes and the player always have the same size and shape of 1×1.
- In Sokoban, the player can only push one box at a time.

3.3.1 Sokoban PDDL

Sokoban can also be represented in PDDL [31]. Because the environment does not allow different shapes and sizes of objects and the player can only push one box at a time, the domain definition is more straightforward than that of PushWorld. It defines the following predicates:

- **`has_player(?x)`:** Indicates that the player is at position `?x`.
- **`has_box(?x)`:** Indicates that a box is at position `?x`.
- **`adjacent(?x, ?y)`:** Indicates that position `?x` is adjacent to position `?y`.
- **`adjacent_2(?x, ?y)`:** Indicates that position `?x` is adjacent to position `?y` with a distance of two.

The grid shape is defined by both `adjacent` and `adjacent_2`. The domain also defines the actions:

- **`move-player(?x, ?y)`:** Moves the player from position `?x` to position `?y`.
- **`push-box(?x, ?y, ?z)`:** Moves the player from position `?x` to position `?y` and the box from position `?y` to position `?z`.

4 Methods

This chapter outlines the methods used for our experiments. We first introduce Restricted-PushWorld, a modified version of Sokoban with normal boxes, along with its variants. We then demonstrate the problem's hardness by providing the idea of a proof that RestrictedPushWorld and one of its variants remain NP-hard like Sokoban. This implies that there is no general policy, especially not an optimal one, for them in our approach. Finally, we describe the modifications made to the existing implementation to accommodate our problem. Most of these changes are necessary because the existing code relies on the complete state spaces of the MDPs, which are too large for our problem. For example, we implemented a replay buffer to store a limited number of states and added a dead-end heuristic, a correct but incomplete strategy to detect unsolvable problems.

4.1 Adapting the Existing Sokoban PDDL

Instead of using the original PushWorld PDDL, we use a modified version of the Sokoban PDDL, because:

1. The implementation of Ståhlberg *et al.* [28] does not support the `:typing`, `:conditional-effects` and `:negative-preconditions` requirements. Although it is possible to omit the `:typing` requirement and bypass the `:negative-preconditions` requirement, the `:conditional-effects` requirement is essential and cannot be removed easily.
2. The implementation also does not support using multiple domains at once. Thus, we cannot train on problems with varying numbers of boxes since the domain depends on the number of boxes.
3. The domain is somewhat complex because it splits up one move into multiple actions. This would likely go beyond the scope of the GNN expressivity.

Except for the `:negative-preconditions` requirement, these issues do not arise in the Sokoban PDDL. The Sokoban PDDL is also more straightforward to understand. However, this limits us to 1×1 shapes and restricts the player to pushing only one box at a time.

To introduce normal boxes (boxes not present in the goal definition) into the Sokoban PDDL, we replace the `has_box` predicate with two new predicates:

- **`has_normal_box(?x)`:** Indicates that a normal box is at position `?x`.
- **`has_goal_box(?x)`:** Indicates that a goal box is at position `?x`.

The `move-player` action remains as is because it does not use the `has_box` predicate, but we must modify the `push-box` action to use the new predicates. Ideally, we want to replace the `has_box` with a disjunction of the new predicates. However, we can not use the `:disjunctive-preconditions` requirement to achieve this because it is not supported. An easy workaround is to duplicate the `push-box` action. In the original, we replace all occurrences of `has_box` with `has_normal_box(?x)`. We do the same in the copy with `has_goal_box(?x)`.

Lastly, we remove the `:negative-preconditions` requirement. For every predicate that was used at least once negated, we introduce a new predicate that depicts the negation of the original one:

- **`not_has_player(?x)`:** Indicates that the player is **not** at position `?x`.
- **`not_has_goal_box(?x)`:** Indicates that a goal box is **not** at position `?x`.
- **`not_has_normal_box(?x)`:** Indicates that a normal box is **not** at position `?x`.

Then, in every effect where we modify the original predicate, we also need to modify the negated one accordingly. The initial state in each problem must also be adjusted to include the negated predicates. This way, we can simulate the negation of predicates without using `:negative-preconditions`. The full modified Sokoban PDDL is shown in Listing A.3.

Although the domain supports multiple goals, due to time constraints, this thesis focuses exclusively on evaluating problems with only one goal. Doing so, we can still compare our results to the original ones of PushWorld from Kansky *et al.* [17], and as we will see, the problem is already hard enough. We call this version of Sokoban RestrictedPushWorld.

4.2 Introducing the Variants

As part of RestrictedPushWorld, we introduce distinct variants:

- **V1:** Player and single goal box (simple navigation)
- **V2:** Player, single goal box, and arbitrarily many walls (obstacle avoidance)
- **V3:** Player, single goal box, and arbitrarily many normal boxes (obstacle manipulation)

The variants are designed to test the expressivity of the GNN and seperate possible scenarios. We also take a look at some Level 0 variants of PushWorld introduced in Section 3.2 in order to compare our results to theirs:

- **PWBASE:** Corresponds to PushWorld's BASE category. So puzzles have a fixed size of 5×5; there is the player, one goal box, one without a goal, and three walls.
- **PWLARGERPUZZLES:** Corresponds to PushWorld's LARGERPUZZLESIZES category. So it is the same as PWBASE, but puzzle sizes uniformly sampled from $5, \dots, 10$.
- **PWMOREWALLS:** Corresponds to PushWorld's MOREWALLS category. So it is the same as PWBASE, with the number of walls sampled from $3, \dots, 5$.

- **PwMoreObstacles:** Corresponds to PushWorld's MoreObstacles category. So it is the same as PwBase, but with two normal boxes.

In addition, we can calculate the number of possible puzzles for each variant as follows, where n is the number of grid cells:

$$\underbrace{n}_{\substack{\text{Possible Player}\\\text{Positions}}} \cdot \underbrace{\binom{n-1}{|\text{Walls}|}}_{\substack{\text{Possible Wall}\\\text{Positions}}} \cdot \underbrace{\binom{n-|\text{Walls}|-1}{|\text{Goal Boxes}|}^2}_{\substack{\text{Possible Goal and}\\\text{Goal Boxes Positions}}} \cdot \underbrace{\binom{n-|\text{Walls}|-|\text{Goal Boxes}|-1}{|\text{Normal Boxes}|}}_{\text{Possible Normal Boxes Positions}} \tag{4.1}$$

We can also estimate the number of possible states for a puzzle. The following provides an upper bound for a puzzle with n grid cells:

$$\underbrace{(n-|\text{Walls}|)}_{\substack{\text{Possible Player}\\\text{Positions}}} \cdot \underbrace{\binom{n-|\text{Walls}|-1}{|\text{Goal Boxes}|}}_{\text{Possible Goal Boxes Positions}} \cdot \underbrace{\binom{n-|\text{Walls}|-|\text{Goal Boxes}|-1}{|\text{Normal Boxes}|}}_{\text{Possible Normal Boxes Positions}} \tag{4.2}$$

We implemented a simple generator that creates random problem instances for each variant based on the grid dimensions, the number of normal boxes, and the number of inner walls. Although we focus on square grids, the generator also supports rectangular ones. To ensure solvability, we use LAMA [25], an off-the-shelf planner that builds on the Fast Downward system [13], to find a solution. They are generally obtained in less than 3 minutes, most being optimal or near-optimal. For simplicity, we consider the solution found to be optimal. The plans are saved along the problems and will be used to compare the performance of our approach to the optimal solution. We also use the plans for validation, which we explain later in Section 4.4.2. The generator is easily extendable. For example, it is possible to impose restrictions on the generated problems, such as the optimal plan length, properties of the optimal plan, or the position of certain boxes, the player, or the goal.

4.3 Hardness of the Problem

As soon as we limit the grid size, the problem is in P because there are only a finite number of possible puzzles and states. This applies to all PushWorld variants PwBase, PwLargerPuzzles, PwMoreWalls, and PwMoreObstacles, meaning that they are in P. Limiting the number of boxes also places the problem in P because then the number of possible states grows polynomially with respect to the grid size. We can confirm this by estimating an upper bound for Equation (4.2):

$$\underbrace{n}_{\substack{\text{Possible Player}\\\text{Positions}}} \cdot \underbrace{n^{|\text{Goal Boxes}|}}_{\substack{\text{Possible Goal}\\\text{Boxes Positions}}} \cdot \underbrace{n^{|\text{Normal Boxes}|}}_{\substack{\text{Possible Normal}\\\text{Boxes Positions}}} = n^{1+|\text{Goal Boxes}|+|\text{Normal Boxes}|} \tag{4.3}$$

Problem	Class	Reason
SOKOBAN	NP-hard	Among others proven by Dor and Zwick [5]
RESTRICTEDPUSHWORLD	NP-hard*	Shown in Section 4.3.3
V1	P	Fixed number of boxes (none)
V2	P	Fixed number of boxes (none)
V3	NP-hard*	Shown in Section 4.3.4
PWBASE	P	Fixed grid size and number of boxes
PWLARGERPUZZLES	P	Fixed grid size and number of boxes
PWMOREWALLS	P	Fixed grid size and number of boxes
PWMOREOBSTACLES	P	Fixed grid size and number of boxes

Figure 4.1: An overview of the hardness of generally determining if a problem instance is solvable and its reason for each variant. *We only provided the idea of a proof.

Since $1 + |\text{Goal Boxes}| + |\text{Normal Boxes}|$ is constant when limiting the number of boxes, Equation (4.3) grows polynomial in relation to n. Thus, Equation (4.2) and the number of possible states both grow polynomially with respect to the grid size. Therefore, V1 and V2 also are in P.

For V3, the situation is more complicated. While we know that Sokoban is NP-hard [5], it is unclear if this property remains when normal boxes are introduced, and the problem is limited to a single goal box. This is because the NP-hardness proof for Sokoban depends on multiple goal boxes. Moreover, it is uncertain whether the problems stay NP-hard when removing walls entirely, bringing us to V3. This section sketches the idea of a proof that RestrictedPushWorld and V3 remain NP-hard, by following the same idea as the proof of original Sokoban from Dor and Zwick [5], with some modifications. We will give an idea how to reduce the RestrictedPushWorld to a known NP-hard problem and then demonstrate that V3 would suffice for the same proof. Figure 4.1 provides an overview of the hardness and its reason for each variant.

4.3.1 Monotone Linked Planar 3-SAT

One famous NP-hard problem is the boolean satisfiability problem (SAT). In SAT, we are given a boolean formula in conjunctive normal form (CNF), and we have to decide whether there is an assignment of the variables such that the formula evaluates to true. It can be shown that the property of NP-completeness remains even if we restrict the boolean formula to have precisely three literals per clause, which is known as 3-SAT.

SAT (NP-Complete)

Input: A boolean formula ϕ in CNF.
Question: Is there an assignment of the variables such that ϕ evaluates to true?

3-SAT (NP-Complete)

Input: A boolean formula ϕ in CNF where each clause has exactly three literals.
Question: Is there an assignment of the variables such that ϕ evaluates to true?

Let ϕ be a boolean formula in CNF, C be the set of clauses in ϕ, and V be the set of variables in ϕ. Then the graph $G_\phi = \langle V \cup C, E \rangle$ is defined with $E = \{(v, c) \in V \times C \mid v \text{ or } \neg v \text{ appears in } c\}$. A graph is planar if it can be drawn in the plane without any crossing edges. It can be shown that 3-SAT remains NP-hard even if G_ϕ is restricted to be planar [19, 24].

PLANAR 3-SAT (NP-Complete)

Input: A boolean formula ϕ in CNF where each clause has exactly three literals, and the planar graph G_ϕ.
Question: Is there an assignment of the variables such that ϕ evaluates to true?

A clause $c \in C$ is monotone if it contains either only positive or negative literals. It can be shown that PLANAR 3-SAT remains NP-hard even if the formula is further restricted to only have monotone clauses [24]. Additionally, G_ϕ can be further restricted to have a Hamilton cycle κ of $C \cup V$ that visits first all variables and then all clauses [24], which is called linked.

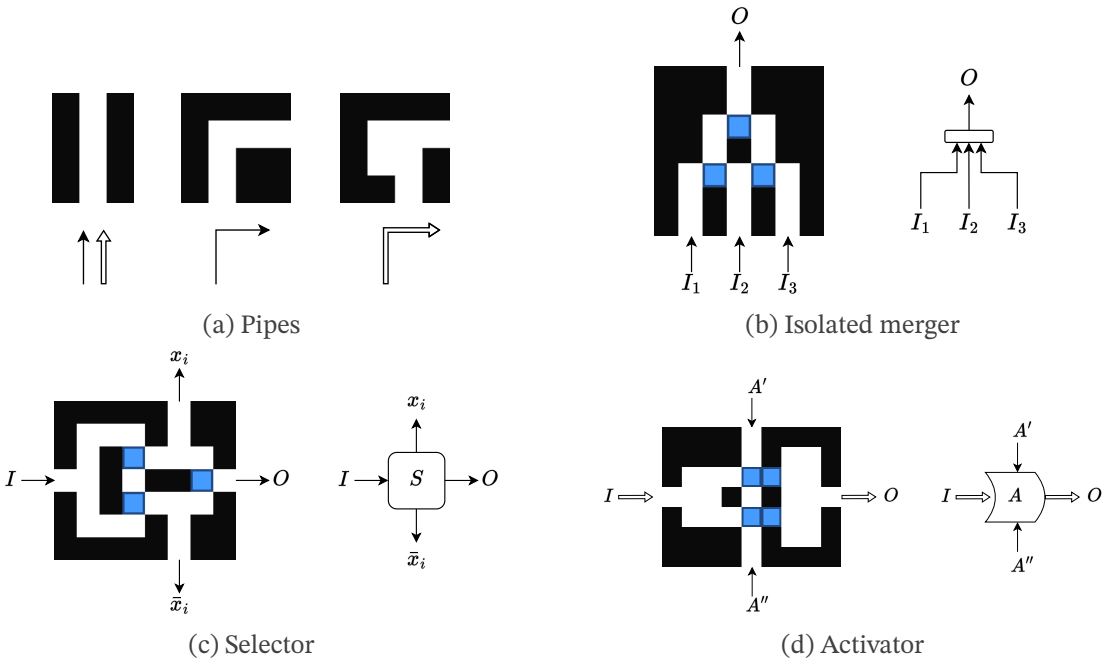
MONOTONE LINKED PLANAR 3-SAT (NP-Complete)

Input: A boolean formula ϕ in CNF where each clause is monotone and has exactly three literals, the planar graph G_ϕ, and a Hamilton cycle κ of $C \cup V$ that first visits all variables and then all clauses.
Question: Is there an assignment of the variables such that ϕ evaluates to true?

4.3.2 Modules

First, some modules need to be introduced. A normal arrow (→) represents a path not intended to be traversed with a box. If the agent tries to push a box along this path, the box typically gets stuck. The arrow indicates the intended direction for the agent's movement, though it may also move in the opposite direction. A double arrow (⇨) represents a path designed to be traversed with a box, but only in the specified direction. However, the agent can still move in both directions without a box. Both path types, and corresponding pipes are shown in Figure 4.2a.

The first module is the isolated merger shown in Figure 4.2b. The agent can enter from I_1, I_2, or I_3 and either exit through O or the input from which it has entered. Exiting through an input that has not been used before is impossible. Traversing the module with a block is also not possible.



(a) Pipes (b) Isolated merger

(c) Selector (d) Activator

Figure 4.2: Modified RestrictedPushWorld modules alongside their symbols

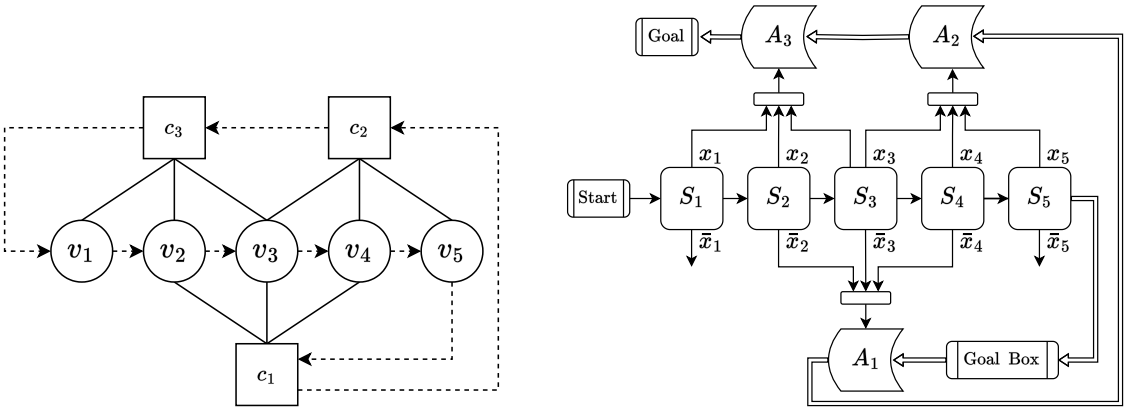
The second module is the selector shown in Figure 4.2c. The agent is assumed to enter from I. If it wants to proceed, it has to decide whether to unlock path x_i or $\overline{x}_i$. This decision is irreversible. Afterward, the agent can exit through O. It is impossible to traverse the module with a block.

The third module is the activator shown in Figure 4.2d. When the agent enters while pushing a block from I, it can only exit through O if it has unlocked path A' or A'' beforehand and pushed the blocks in the middle out of the way. This means that the selector has two possible states: locked or unlocked. Pushing blocks in the opposite direction from O to I is impossible.

4.3.3 Reduction Sketch

First, we will show that RestrictedPushWorld can be reduced to MONOTONE LINKED PLANAR 3-SAT. Specifically, we show that we can encode any MONOTONE LINKED PLANAR 3-SAT problem as a RestrictedPushWorld problem such that a solution corresponds to a solution to the MONOTONE LINKED PLANAR 3-SAT problem. Thus, we must provide a way to transform any boolean formula ϕ in CNF where each clause is monotone and has precisely three literals to a RestrictedPushWorld problem given the planar graph G_ϕ, and a Hamilton cycle κ of $C \cup V$ that first visits all variables and then all clauses. An example of this reduction is shown in Figure 4.3, which can be used to follow the coming explanation.

For each variable $v_i \in V$, we use a selector corresponding to unlocking path x_i if v_i is true and $\overline{x}_i$ if v_i is false. For each clause $c_j \in C$, we use an activator A_j. If A_j is unlocked, then c_j is evaluated as true. Each section x_i is connected to each c_j if x_i appears in c_j according to G_ϕ.



(a) A monotone linked planar graph G_ϕ (b) Its corresponding RestrictedPushWorld problem

Figure 4.3: Example of a monotone linked planar graph G_ϕ and the corresponding RestrictedPushWorld problem of $\phi \doteq (\overline{x}_2 \vee \overline{x}_3 \vee \overline{x}_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_4 \vee x_5)$.

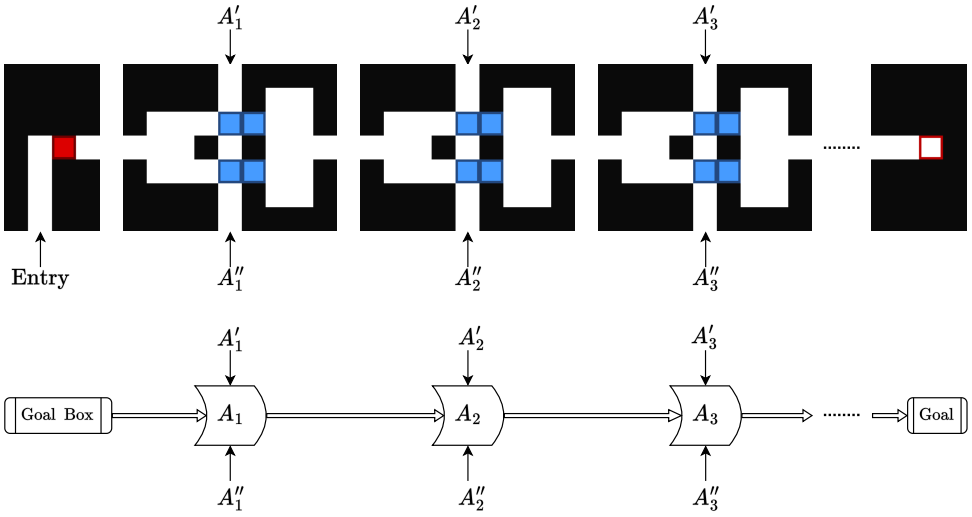
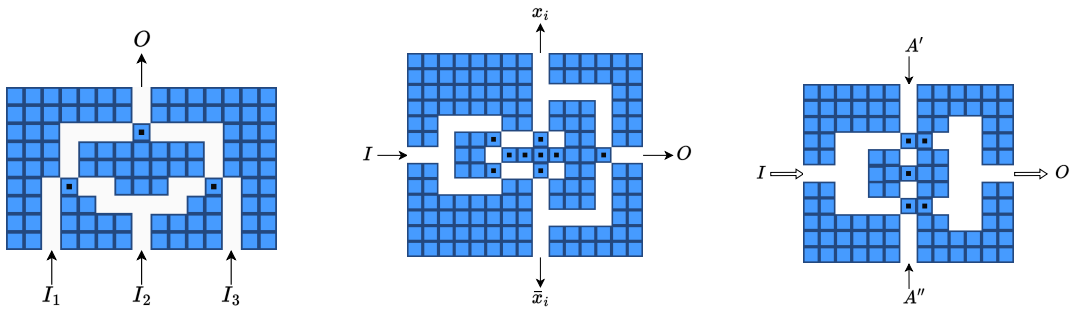


Figure 4.4: Assembled goal chain alongside its symbols

The same goes for $\overline{x}_i$. Before reaching the activator, all paths that lead to it are merged by an isolated merger to ensure that it does not skip anything. Since G_ϕ is planar, this construction is possible without any edges crossing.

The Hamilton cycle κ is used to connect first all variable selectors and then all clause activators. The latter results in the goal chain shown in Figure 4.4. The goal box can only be pushed to the goal if all activators are unlocked, only if the choices at the selector modules are correct. The construction of the RestrictedPushWorld problem is possible in polynomial time, given G_ϕ and κ. Since a solution to the RestrictedPushWorld problem corresponds to a solution to the MONOTONE LINKED PLANAR 3-SAT problem, RestrictedPushWorld is also NP-hard.



(a) Isolated merger (b) Selector (c) Activator

Figure 4.5: Modified RestrictedPushWorld modules from Figure 4.2 without walls

4.3.4 Removing Walls

From here on, it is easy to prove that V3, so RestrictedPushWorld without any walls, should remain NP-hard. The key is to exploit the fact that the agent can only move at most one block at a time. Thus, walls must be replaced with immovable normal boxes. It suffices to show that all modules can be constructed without any walls. The updated modules are shown in Figure 4.5. To clarify, theoretically, movable boxes are marked with a black dot (■), and boxes that could never be moved remain unmarked (■).

The alleged NP-hardness of V3 and RestrictedPushWorld means there is no general policy, especially not an optimal one, for them in our approach. Nevertheless, we still can aim for a policy that is as good as possible. Like Ståhlberg *et al.* [28], our focus lies on generalization, finding a policy that works well for as many problems as possible. Even small randomly created puzzles can become quite challenging. Figure A.1 and Figure A.2 show a collection of such puzzles.

4.4 Adapting the Existing Implementation

The existing implementation from Ståhlberg *et al.* [28] is unsuitable for our problem as it relies on the complete state spaces of the MDPs. This section details the necessary changes needed to make it work for our problem.

4.4.1 Replay Buffer

The generated state spaces are used to obtain uniform state samples, perform validation, and detect dead-ends. However, this approach of generating whole state spaces is impractical for our problem due to the large state spaces. Instead, we use a replay buffer to only store a limited number of states. Normally, a replay buffer stores tuples of states, actions, rewards, next states, and next actions: $\langle s, a, r, s', a' \rangle$. But since we can generate actions and successors at any time

Algorithm 3 Replay Buffer Fill

Input: MDPs $\{M_1, \dots, M_N\}$, maximal replay buffer size K, and maximal trace length T

function APPEND(ordered_set, element)
    Append element to ordered_set if it is not already in it
    Terminate filling if $K \leq |\text{ordered\_set}|$
function POP(ordered_set)
    Remove and return the first element of ordered_set

Initialize replay buffer $R \leftarrow \emptyset$ and helper set $S \leftarrow \emptyset$ (both are ordered by insertion time)
for all $M \in \{M_1, \dots, M_N\}$ **do**
    for all $s_0 \in$ initial states of M **do**
        APPEND(R, s_0) and APPEND(S, s_0)

while $0 < |S|$ **do**
    $s \leftarrow$ POP(S)
    for $i = 1, \dots, T$ **do**
        $S' \leftarrow \{s' \in N(s) \mid s' \notin R\}$
        if $S' = \emptyset$ **then**
            break (only from inner loop)
        for all $s' \in S'$ **do**
            APPEND(R, s') and APPEND(S, s')
        $s \leftarrow$ uniformly sample from S'

by consulting the underlying model, and rewards are always -1, the replay buffer only needs to store states.

Before training, the replay buffer gets filled using random traces using Algorithm 3. It is ensured that no duplicate states are added. During training, newly discovered states are added to the replay buffer while others are removed to make space. There are several ways to decide which states to sample and which to remove. We use a simple first-in-first-out (FIFO) strategy because it is easy to implement and works well in practice. It ensures that the replay buffer always contains the most recent states.

4.4.2 Validation

The state spaces are also necessary for validation. Before training, the optimal state value function v_* is computed for each MDP. During training, at each validation step, the state value function of the current policy v_{π_θ} for every MDPs is computed using policy evaluation requiring all states. The iterative policy evaluation terminates when the update falls below a predefined threshold. The validation loss is the sum of the value function of each state weighted by the number of states in each MDP shown in Equation (4.4). We also know the optimal validation loss shown in Equation (4.5) and can terminate the training when their difference drops below a certain threshold. We call this currently implemented method exact validation.

$$\mathcal{L}_{\text{val}}^{\text{exact}} \doteq \sum_{\text{MDP}} \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} v_{\pi_\theta}(s) \qquad (4.4) \qquad\qquad \mathcal{L}_{\text{val}*}^{\text{exact}} \doteq \sum_{\text{MDP}} \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} v_*(s) \qquad (4.5)$$

Instead of splitting the supplied MDPs into training and validation sets beforehand, we have a dedicated validation set. Its MDPs are either small enough to generate the complete space space or find an optimal plan by an off-the-shelf planner. This approach enables exact validation for the validation MDPs with small enough state space. We introduce another method called policy search evaluation for the MDPs with larger state spaces. Prior to training, we use an off-the-shelf planner to find plans. During training, we simulate the current policy for a maximal number of steps and calculate the ratio of the achieved plan length to the optimal. Though less accurate, policy search evaluation still provides a reasonable estimate. If the policy does not solve the problem, we use a predefined punishment factor $\rho \in \mathbb{R}$. The validation loss can now be defined as shown in Equation (4.6), and the optimal validation loss in Equation (4.7).

$$\mathcal{L}_{\text{val}}^{\text{policy search}} \doteq \sum_{\text{MDP}} \frac{1}{\text{OptimalPlanLength}} \cdot \begin{cases} \text{FoundPlanLength} & \text{if solved} \\ \rho \cdot \text{Horizon} & \text{if unsolved} \end{cases} \qquad (4.6)$$

$$\mathcal{L}_{\text{val}*}^{\text{policy search}} \doteq |\text{MDP}| \qquad (4.7)$$

Finally, we need to combine the exact and policy search validation losses. Both losses are normalized by their corresponding optima. We also introduce two factors $\sigma_1, \sigma_2 \in \mathbb{R}^+$ to balance them, compensating for the number of MDP used to compute them. If we assume that we used n^{exact} MDPs for the exact validation and $n^{\text{policy search}}$ MDPs for the policy search validation, we can set $\sigma_1 = \frac{n^{\text{exact}}}{n^{\text{exact}}+n^{\text{policy search}}}$ and $\sigma_2 = \frac{n^{\text{policy search}}}{n^{\text{exact}}+n^{\text{policy search}}}$, and define the combined validation loss as:

$$\mathcal{L}_{\text{val}} \doteq \sigma_1 \frac{\mathcal{L}_{\text{val}}^{\text{exact}}}{\mathcal{L}_{\text{val}*}^{\text{exact}}} + \sigma_2 \frac{\mathcal{L}_{\text{val}}^{\text{policy search}}}{\mathcal{L}_{\text{val}*}^{\text{policy search}}} \qquad (4.8)$$

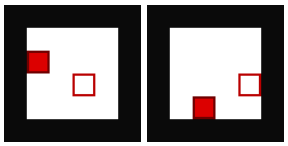
4.4.3 Dead-End Heuristic

Complete state spaces also help to detect dead-ends. This information is used during training to ensure that the value function is infinite for dead-end states $s_\times$, or rather that $v(s_\times) = \frac{1}{1-\gamma}$ since we use a discounting factor $0 < \gamma < 1$ with a fixed action cost of 1.

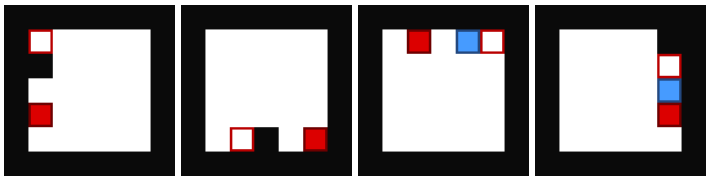
However, detecting dead-ends is infeasible in our case because determining if a state is a dead-end is as hard as determining if a problem is solvable, which we assume to be NP-hard for RestrictedPushWorld. Nonetheless, we can find easily detectable patterns that guarantee a state is a dead-end. This method is correct but incomplete, so we call it a dead-end heuristic.

We introduce the following rules to detect dead-ends with minimalistic examples where the agent's position is irrelevant and thus not shown:

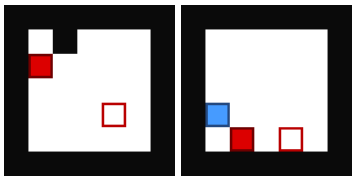
1. If the goal box is positioned on an outer wall and the goal itself is not located along the same outer wall, the state is a dead-end.



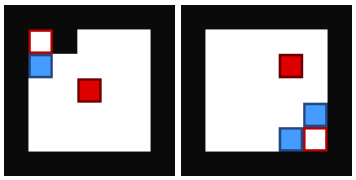
2. If the goal box and the goal are positioned along the same outer wall with another wall between them, or if they are on the same wall with a normal box between them and a wall "behind" the goal, the state is a dead-end.



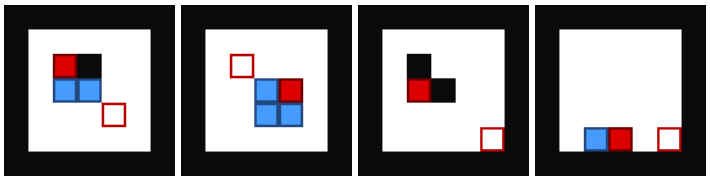
3. If the goal box is adjacent to a corner and there is a normal box or wall diagonally positioned from it, the state is a dead-end unless the goal or the player is positioned exactly in that corner. Note that the player being in this corner is only possible in the initial state.



4. If the goal is positioned in a corner and the two adjacent cells are normal boxes or walls, the state is a dead-end.



5. If the goal box is part of a square pattern involving normal boxes or walls, this time also including outer walls, the state is a dead-end. Similarly, if the goal box has two adjacent walls placed diagonally of it, the state is also a dead-end.



All these patterns, except the second one, evaluate in constant time and cover most cases seen in practice. In the worst case, pattern two scales with the grid width or height. If the grid has dimensions $m \times n$, it scales with $\max\{m, n\}$, but in practice, it is still more than fast enough because the first check if the goal box and the goal are both located along the same wall is constant. The downside of the dead-end heuristics approach is that finding such patterns depends on the domain and cannot be generalized.

4.4.4 Confident Actor But Incorrect Value Function

Another encountered issue is that the actor head learns too quickly. After just a few iterations, its probabilities of selecting a successor are either near 100% or 0%. The value function head does not learn at all, and the values are entirely off. One possible issue we suspect is that because the actor learns so quickly, it no longer explores, leaving the training stuck on the small fragment it has chosen. There are the following approaches to address this issue:

Spliting the Optimizers

The current implementation uses one Adam optimizer [18], which has the same learning rate for all parameters. Instead, we can split the parameters of the actor and critic heads into two optimizers, allowing us to supply two different learning rates. In our case, we need to lower the learning rate of the actor head and/or increase the one of the value function head.

ϵ-Policy

Another approach is introducing a small $\epsilon \in [0, 1]$ [30]. In training, with a chance of ϵ, we pick a uniformly random action instead of following the policy. This way, we can ensure the training is not stuck on a small fragment. We decrease ϵ over time to let the policy converge. After each epoch or batch we multiple ϵ by a decay factor $\epsilon_{\text{decay}} \in [0, 1]$.

Entropy Regularization

Entropy regularization is another method to encourage exploration [21]. The entropy of the policy output is defined as:

$$H(\pi(\cdot \mid s)) = -\sum_{s' \in N(s)} \pi(s' \mid s) \log \pi(s' \mid s) \tag{4.9}$$

We can add the entropy to the loss function with a factor $H_{\text{factor}} \in \mathbb{R}^+$ that controls the strength of the regularization. This factor also needs to be decreased over time, which is done by a decay factor $H_{\text{decay}} \in [0, 1]$ after each epoch or batch. We did not achieve good results with this method, which is why we will not use it in the following experiments, but it is still worth mentioning.

5 Experiments

In this chapter, we detail the experiments conducted to evaluate the performance of the trained models. We describe the setup and data used, and then present the experimental results. This includes initial observations on the models' behavior on the test sets and identifying potential issues. Differences between models trained with and without the dead-end heuristic are noted during the evaluation. Subsequently, we conduct a deeper analysis of specific problem tests to provide a more precise and detailed understanding of identified issues.

5.1 Setup and Data

We trained all models on NVIDIA A10 GPUs and 2.30GHz Intel Xeon Platinum 8352M CPUs for up to 48 hours or 20 epochs. The GNN hyperparameters are $k = 64$ and $L = 30$. The discount factor is $\lambda = 0.999$, and the replay buffer is filled with at most 50,000 states for V1 and 200,000 states for others. Algorithm 3 fills the replay buffer with a maximal trace length of 350. For exact validation, we use a maximum space size of 100,000 states. For policy search validation, we apply the factor $\rho = 3$ when no solution is found within 100 steps. Due to the high variance in results, we train five models for each configuration while benchmarking the top five epochs according to validation against test suites and only report the best results achieved.

In addition, we also tested combinations of hyperparameters for the learning rates of actor and critic, for the ϵ-policy and its decay, and enabled/disabled the dead-end heuristic. The primary criterion for evaluation was the coverage of V3 problems. Generally, the coverage decreases when introducing an ϵ, or when increasing the critic's learning rate. Conversely, lowering the actor's learning rate helps to prevent an overly confident policy. Values ranging from $\alpha = 0.000002$ to $\alpha = 0.0002$ worked best. In most cases, the dead-end heuristic improved the coverage slightly. In a few cases, it decreased it. Table A.1 provides more details on the tried hyperparameter combinations. We take a closer look at the configuration with $\alpha = 0.000015$, $\beta = 0.0002$ and $\epsilon = 0$, once with and once without the dead-end heuristic.

Although the best coverage on V3 was achieved with the default parameters ($\alpha = 0.0002$, $\beta = 0.0002$) and dead-end heuristic, these policies are always 100% certain, leaving no room for stochastic outcomes. Moreover, the 2-5% difference in coverage is not significant enough to conclude that its learning rates are superior.

There are a few options for how to run the benchmarks. The policy search can be deterministic or stochastic, meaning that the policy is either greedy or sampled by the given distribution. Additionally, it can be closed or open. In closed mode, revisiting the same state is disallowed. This creates three benchmark modes: closed deterministic, closed stochastic, and open stochastic. Open deterministic is infeasible as revisiting the same state would cause an infinite loop.

The training sets of V1, V2, and V3 contain problems with grid sizes ranging from 4×4 to 10×10. The test sets also include ones of size 15×15. It has also been ensured that V2 and V3 feature problems with varying numbers of walls and normal boxes, occupying 10% to 25% of all cells. All problems were generated randomly and, as discussed, are solvable. For the PushWorld variants, we use the original test sets from Kansky *et al.* [17] converted to the RestrictedPushWorld PDDL definitions, enabling us to compare our results to theirs. Likewise, the training sets match theirs, but we only use the first 250 problems instead of the full 2000. Details on the composition of the sets can be found in Table A.2.

5.2 Results

Table 5.1 shows the outcomes of the models trained with the configuration described above, with and without the dead-end heuristic. We report:

- **Coverage:** the percentage of solved problems.
- **Plan Quality:** the ratio (PQ=PL/OL) of the sum of found plan lengths (PL) to the sum of optimal plan lengths (OL) for all solved problems, indicating how close the found plans are to the optimal ones. A plan quality of 1.0 means that all found plans are optimal, and 1.5, for example, means that they are 50% longer than the optimal ones on average.
- **Validation scores:** for exact and policy search validation.

We present only the best results of closed stochastic, as closed deterministic performs nearly identically, with differences of less than 4%, sometimes better or worse. Open stochastic consistently performs significantly worse than the closed modes. In addition, for V2 and V3, we also show the coverage separated by grid size and density of walls/boxes to evaluate the models' performance concerning these factors. The results are visualized in Figure 5.1 and Figure 5.2.

For the purpose of the following comparison, we refer to the model trained without the dead-end heuristic as Model-1 and the model trained without the dead-end heuristic as Model-2-DEH. Although the models for each variant differ, we use the same naming convention for simplicity, meaning that, Model-1 can refer to the model of V1, V2, ... based on the context.

	Without Dead-End Heuristic (closed stochastic)				
	Train	Test		Validation	
	Coverage	Coverage	PQ	Exact (#)	PS (#)
V1	100% ($^{70}/_{70}$)	100% ($^{60}/_{60}$)	1.55	1.09 (20)	– (0)
V2	69.7% ($^{244}/_{350}$)	54.2% ($^{65}/_{120}$)	1.44	1.43 (60)	– (0)
V3	51.1% ($^{179}/_{350}$)	69.2% ($^{83}/_{120}$)	2.15	1.19 (21)	2.05 (39)
PWBASE	66.0% ($^{165}/_{250}$)	59.5% ($^{119}/_{200}$)	1.20	1.26 (50)	– (0)
PWLARGERPUZZLES	74.8% ($^{187}/_{250}$)	74.0% ($^{148}/_{200}$)	1.64	1.57 (38)	1.57 (12)
PWMOREWALLS	57.2% ($^{143}/_{250}$)	54.5% ($^{109}/_{200}$)	1.21	1.17 (50)	– (0)
PWMOREOBSTACLES	50.0% ($^{125}/_{250}$)	53.5% ($^{107}/_{200}$)	1.40	1.21 (50)	– (0)

	With Dead-End Heuristic (closed stochastic)				
	Train	Test		Validation	
	Coverage	Coverage	PQ	Exact (#)	PS (#)
V1	87.1% ($^{61}/_{70}$)	90.0% ($^{54}/_{60}$)	1.86	1.44 (20)	– (0)
V2	51.3% ($^{154}/_{350}$)	46.7% ($^{56}/_{120}$)	1.50	1.37 (60)	– (0)
V3	74.6% ($^{261}/_{350}$)	69.2% ($^{83}/_{120}$)	2.50	1.08 (21)	1.75 (39)
PWBASE	64.0% ($^{160}/_{250}$)	59.0% ($^{118}/_{200}$)	1.28	1.30 (50)	– (0)
PWLARGERPUZZLES	71.6% ($^{179}/_{250}$)	66.0% ($^{132}/_{200}$)	1.60	2.99 (38)	1.86 (12)
PWMOREWALLS	51.6% ($^{139}/_{250}$)	60.5% ($^{121}/_{200}$)	1.16	1.28 (50)	– (0)
PWMOREOBSTACLES	46.4% ($^{116}/_{250}$)	52.5% ($^{105}/_{200}$)	1.29	1.21 (50)	– (0)

Table 5.1: The coverage and corresponding plan quality on training and test sets. The models from both training and test sets are the same and determined by the best coverage of the test set. Plan quality is defined as the ratio of the sum of found plan lengths to the sum of optimal plan lengths for all solved puzzles. It is only shown for the test set, as the ones from the training set did not provide further insights.
Additionally, validation scores for the selected epoch are presented: "exact" refers to the ratio $\mathcal{L}_{\text{val}}^{\text{exact}}/\mathcal{L}_{\text{val*}}^{\text{exact}}$, and "PS" refers to policy search as ratio $\mathcal{L}_{\text{val}}^{\text{policy search}}/\mathcal{L}_{\text{val*}}^{\text{policy search}}$.
For validation, the number in parentheses (#) indicates the number of problems used for the validation strategy. For coverage, the fraction ($^{\text{num}}/_{\text{total}}$) represents the number of solved problems over the total number of problems.

	PPO		DQN	
	Train Cov.	Test Cov.	Train Cov.	Test Cov.
BASE	88.5%	40.4%	24.9%	19.5%
LARGERPUZZLESIZES	93.2%	71.5%	61.2%	61.8%
MOREWALLS	77.9%	23.3%	18.5%	13.1%
MOREOBSTACLES	64.7%	21.4%	17.6%	13.1%

Table 5.2: The original results from Kansky *et al.* [17] showing the coverage of PPO and DQN on their train and test sets.

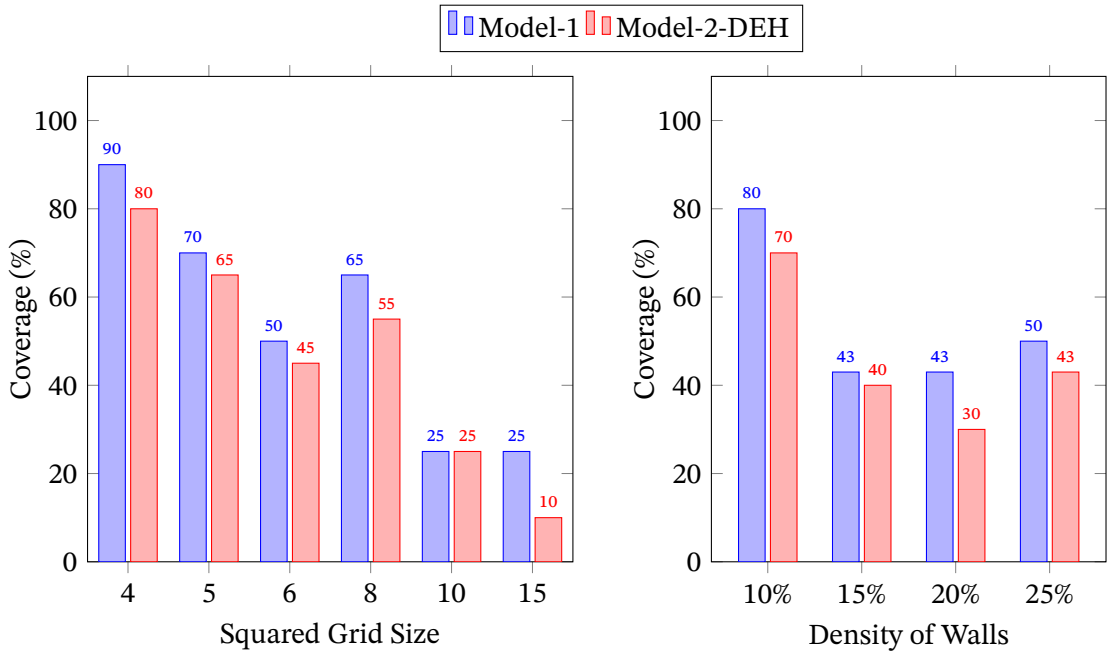


Figure 5.1: The test coverage (closed stochastic) of Model-1 and Model-2-DEH on V2 problems separated by grid size and density of walls. Details on the absolute numbers can be found in Table A.3.

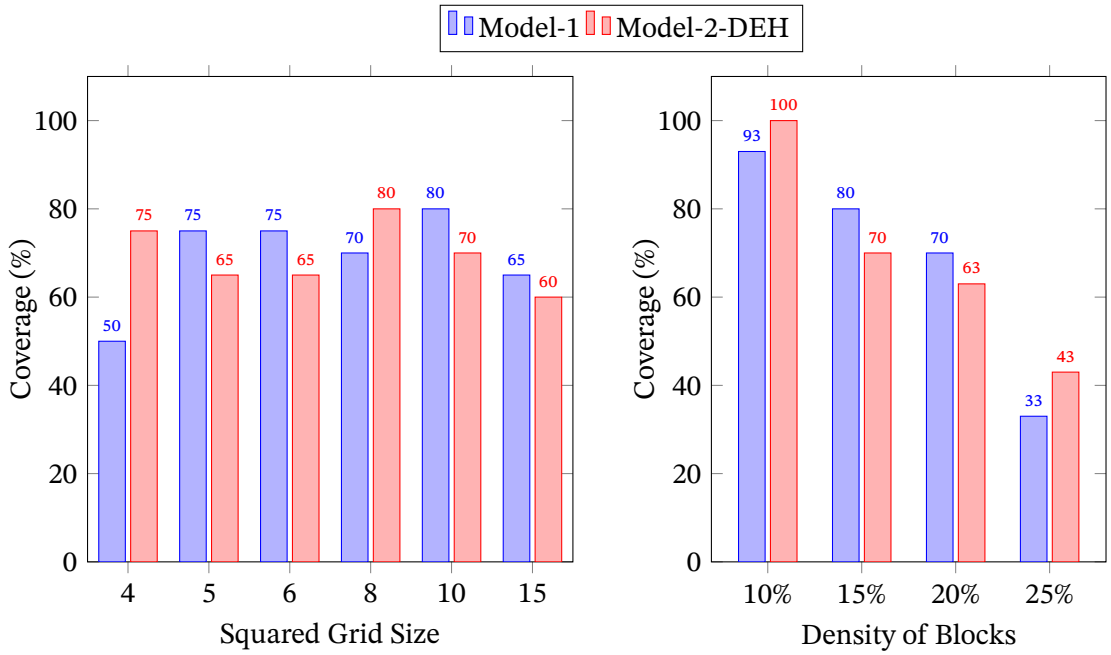


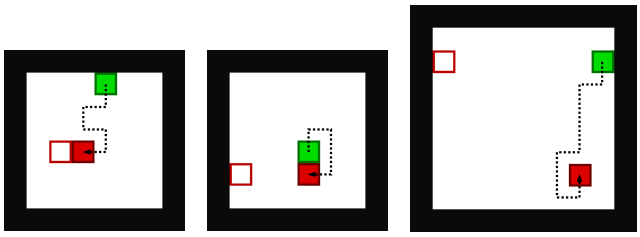
Figure 5.2: The test coverage (closed stochastic) of Model-1 and Model-2-DEH on V3 problems separated by grid size and density of blocks. Details on the absolute numbers can be found in Table A.4.

The following subsections offer initial insights into the models' behavior, identify possible issues, and highlight notable differences between Model-1 and Model-2-DEH. After these initial observations, in the next sections, we conduct more in-depth evaluations of specific problem tests to provide a more precise and detailed understanding of certain issues. All examples of problems and states in this section are taken from the test sets.

5.2.1 V1 Results

Model-1 and Model-2-DEH achieved great results on V1 with test coverages of 100% and 90%, respectively. In the four unsolved problems of Model-2-DEH, the agent had no more unvisited successor states available, and since we use a closed mode, this counts as unsolved. In these problems, the agent did not push the box once and appeared to wander aimlessly. The difference in coverage is likely due to the high variance in training. By training more models, we might also reach 100% coverage with the dead-end heuristic.

As indicated by the plan qualities of 1.55 and 1.86, the found plans are not always optimal. Occasionally, the agent strays slightly from the optimal path, for example:



Another issue, particularly present in larger grid sizes, is that the agent tends to wander around aimlessly when the goal box is not nearby. In these situations, the policy assigns nearly equal probabilities to all possible successors, and the value function reflects this uncertainty by assigning nearly equal values to all possible successors. Once the agent discovers the goal box, it pushes it randomly in different directions until the goal is close enough.

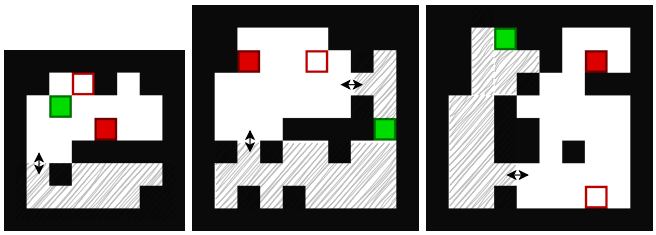
An unusual behavior, observed only in plans of Model-2-DEH, is that the agent sometimes appears to perform well at first, pushing the goal box in the correct direction but then suddenly deviates and wanders away from it. In these instances, it seems that an improbable successor was chosen, and due to the closed mode, the agent cannot return. Thus, the agent is repelled from the goal box. Nevertheless, the open stochastic mode, which allows revisiting states, still has lower coverage because it often enters endless cycles.

5.2.2 V2 Results

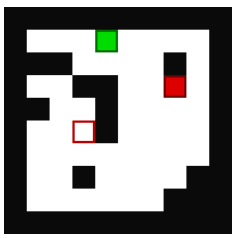
On V2, the test coverages for Model-1 and Model-2-DEH were 54.2% and 46.7%, respectively. As the grid size increases, coverage declines for both models. Model-1 consistently outperforms or matches Model-2-DEH, with minor differences. However, a notable exception occurs on the largest grid size tested (15×15), where Model-2-DEH performs substantially worse. Analysis of the plans produced by Model-2-DEH reveals that the agent again wanders aimlessly, especially when the goal box is not nearby. This issue is more prominent on V2 than on V1.

Examining the coverages based on wall density, both models show a notable drop from 10% to 15% density. Beyond that, coverage remains relatively stable. At 25% density, however, there is an unexpected increase in coverage for both models. Reviewing the problems with 25% density, we observed that they are generally more straightforward to solve than those with 15% and 20% density due to shorter plan lengths and less need to navigate around walls. It appears that generating puzzles with such high densities is unlikely to create complex problems, as most randomly created ones are unsolvable and thus discarded.

In numerous problems, the agents also fail because of the closed mode, where, especially in V2, it is easy for the agent to enter regions or corners from which it cannot return. Also, there are areas with just a few exits, which easily trap the agent. Since our model does not know the previously visited states, it cannot explicitly avoid such situations. The following states feature such regions, which have been shaded, and all exits are marked with arrows:



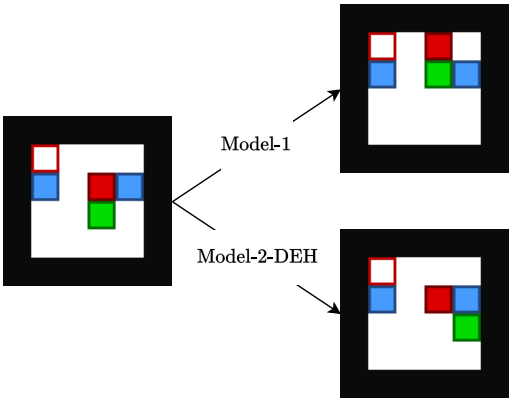
Another issue arises when a wall blocks the agent from pushing in the direction of the goal or when a wall obstructs the direct path between the goal box and the goal. For instance, Model-1 particularly struggles with the following problem, where both challenges are present:



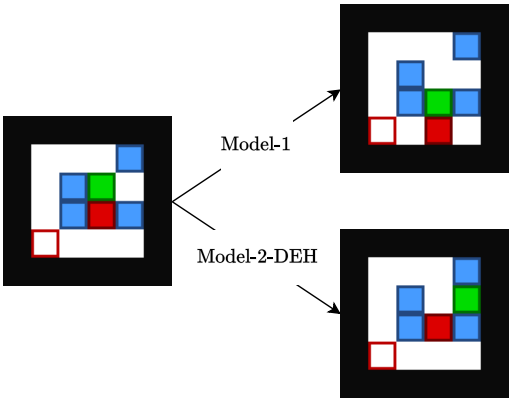
5.2.3 V3 Results

On V3, Model-1 and Model-2-DEH achieved an identical test coverage of 69.2%. The plan quality is 2.5, which is the highest of all variants. Unlike V2, coverage on V3 remains stable across different grid sizes, and both models perform similarly on all grid sizes, except for the smallest, where Model-2-DEH clearly outperforms Model-1. Another difference compared to V2 is that the coverage consistently decreases with block density, lacking the previous increase at 25% density. In contrast to V2, randomly created problems for V3 have a greater chance to be solvable as normal boxes are moveable.

While both models occasionally fail due to closed mode or overly complex puzzles, we observed that Model-1 encounters dead-ends that are detailed in Section 4.4.3. In contrast, Model-2-DEH tends to avoid such situations. For instance, both models encountered the following states:



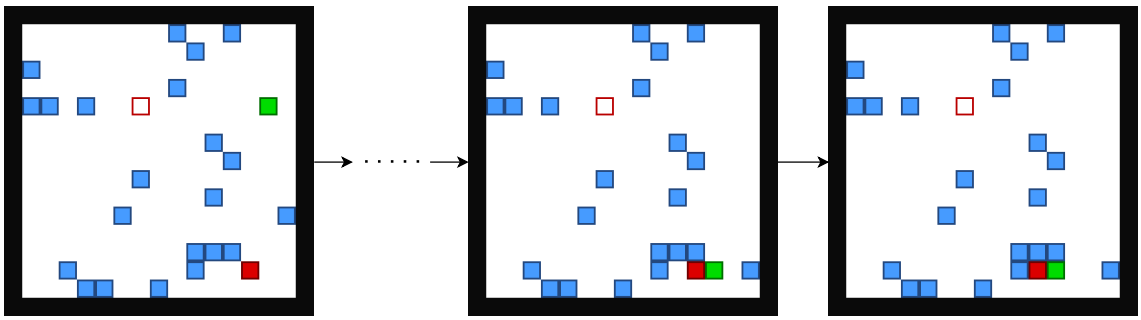
(a) State 1 and selected successors



(b) State 2 and selected successors

In the first state, Model-1 pushed the box up with 100% confidence. Its value function also identified that this is the best move, assigning the chosen successor a lower value than other possible successor states. By doing this, it entered a dead-end, which would be classified as such by pattern three of Section 4.4.3. Conversely, Model-2-DEH moved right, avoiding the dead-end. Its probability of moving up was 0%, and the value function correctly indicated that moving up is incorrect by assigning nearly the highest possible value. Ultimately, Model-2-DEH solved the problem. A similar situation occurred in the second state; both models acted with 100% confidence and made the best move according to their value functions. However, in this case, after getting the goal box closer to goal, Model-2-DEH failed to solve the problem by running into a corner with no unvisited successor states left. From then on, the problem remained theoretically solvable, unlike the state where Model-1 ended up.

Another problem that highlights a difference in the models' behavior is the following, where Model-1 runs into a dead-end:



Again, this decision is made with 100% confidence. Note that pattern five of Section 4.4.3 classifies this dead-end. In the beginning, Model-1 rapidly navigated to the goal box, while Model-2-DEH failed to do so and instead wandered around its starting position, similar to the behavior sometimes observed on V1 and V2. Again, its policy assigns nearly equal probabilities to successors, and the value function reflects this uncertainty, leaving the agent unsure of its next move.

While Model-2-DEH appears to avoid specific dead-ends more than Model-1, it does not improve coverage and fails to solve problems for other reasons. We must further analyze whether Model-2-DEH truly understands dead-end situations better, as the limited examples in the test set are insufficient to draw a definitive conclusion.

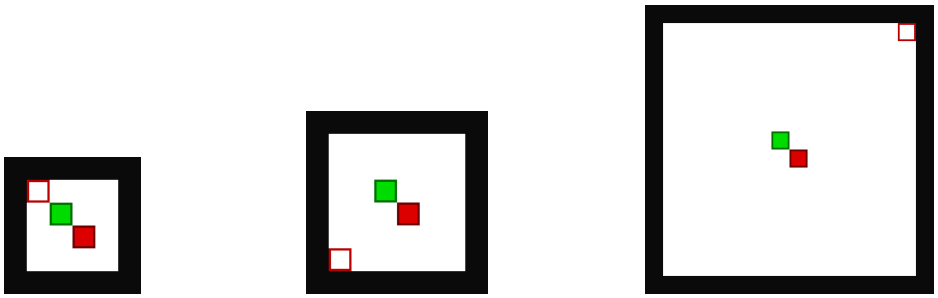
5.2.4 PushWorld Variants Results

Model-1 and Model-2-DEH perform similarly on all PushWorld variants. Since we use the same test sets as Kansky *et al.* [17], we can directly compare our results to theirs. In PWBASE, PWMOREWALLS, and PWMOREOBSTACLES, our approach significantly outperforms their results with PPO and DQN on the test sets shown in Table 5.2. However, in PWLARGERPUZZLES, our approach performs slightly worse than DQN but better than PPO, though the difference is minor. The coverage of our models is more stable across the different variants than the coverage of PPO and DQN.

Kansky *et al.* [17] faced substantial differences between train and test coverage. In contrast, our approach shows greater robustness in this regard, with minimal differences between them, sometimes even favoring the test coverage. This also applies to the other variants V1, V2, and V3, suggesting that our models generalize well to unseen problems. Besides this, PWLARGERPUZZLES has a particularly higher coverage as the other variants in the results of Kansky *et al.* [17], whereas the difference is not as clear in our results. Although, PWLARGERPUZZLES also has the highest coverage, the difference to the other variants is not as significant.

5.3 Special Tests

After the initial observations made in the previous section, we conduct more in-depth evaluations of specific problem tests to provide a more precise and detailed understanding of certain issues. Unlike above, we now evaluate with closed deterministic mode to ensure that the agent chooses the most likely action.



(a) (4 × 4, 9 steps) (b) (6 × 6, 8 steps) (c) (14 × 14, 16 steps)

Figure 5.4: Examples of V1-INCREASING problems with their grid size and optimal plan length

5.3.1 Increasing Puzzle Sizes

We often encountered issues where the agent wandered in wrong directions when the goal box was not nearby, suggesting that the models struggle to identify the correct direction over larger distances. To investigate this further, we created problems where the agent and the goal box are positioned in the center, and the goal is located at a corner. We then analyze whether the agent moves in the correct direction, defined as the direction to the two adjacent walls of the goal, across increasing squared grid sizes. The agent fails the test if it pushes the goal box at least once in the wrong direction, meaning that the distance between goal box and goal increases, or if the agents starts to wander aimlessly. The problems are called V1-INCREASING.

Besides the limited expressivity of GNNs, they are also limited by the hyperparameters k and L. Precisely, the number of message passing steps L is crucial for the GNN to learn the correct direction. Ignoring walls or normal boxes, GNNs cannot determine path existence between two cells if the distance exceeds $2L$. Furthermore, the cells cannot be more than L steps away if the full path is required, which is needed to identify the direction in which the box needs to be pushed.

The results are shown in Figure 5.5. Both models achieved a coverage of 100%, but their number of passed test differs significantly. Model-1 passed all tests up to a grid size of 9×9, corresponding to a distance of seven to nine cells between goal box or agent to the goal. In contrast, at the smallest grid size of 4×4, Model-2-DEH began to make errors, by pushing the goal box in the wrong direction. From grid size 8×8 onward, it failed all tests.

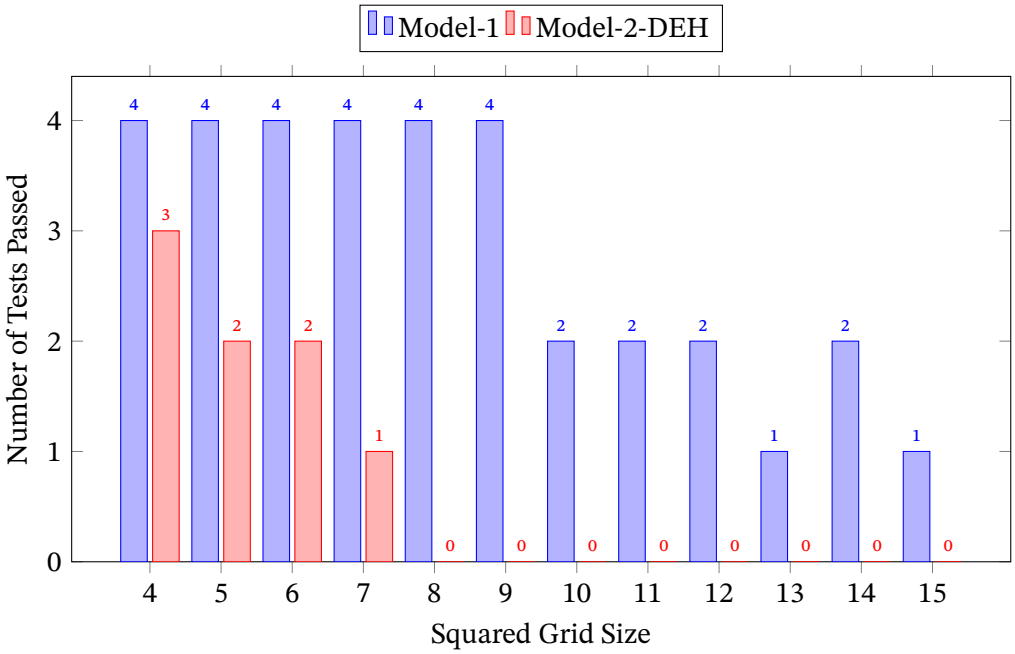


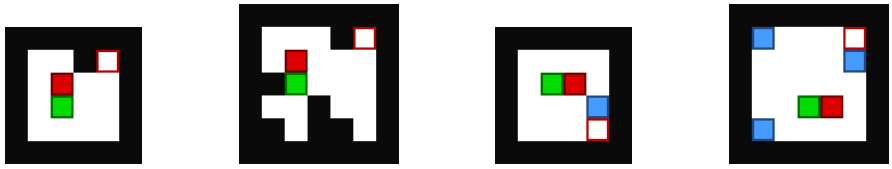
Figure 5.5: The number of problems passed by Model-1 and Model-2-DEH on V1-INCREASING separated by grid size closed deterministic. There are a total of four problems per grid size.

Besides these tests, it would be interesting to observe how the models perform when the goal box is not placed in the center with the agent but besides the goal. This would test how the models behave when the agent moves alone. Additionally, adding other normal boxes could test if they make a difference. These tests could provide further insights into the models' behavior and limitations, but they will not be included in this thesis due to time constraints.

5.3.2 One-Step Dead-Ends (OSDEs)

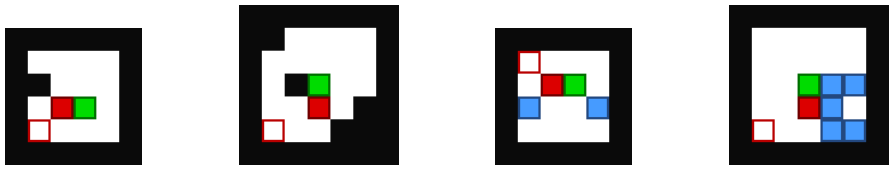
As mentioned in the analysis above, it seems that Model-1 goes into dead-ends more often than Model-2-DEH. To investigate this further, we created several test sets in which the agent is only one step away from a dead-end. This also provides inside if the GNN can identify and avoid the specific patterns of dead-ends.

We created problems to test pattern two of Section 4.4.3. In these scenarios, the agent is always one step away from pushing the goal box towards an edge where the goal is located. If the agent chooses to do so, it enters a dead-end for one of two possible reasons: there is a wall or normal box between the goal box and the goal, or there is a wall or normal box next to the goal box on the other side. The problems corresponding to the first case are called V2-OSDE-P2-1 and V3-OSDE-P2-1, while those for the second case are V2-OSDE-P2-2 and V3-OSDE-P2-2. We differentiate the cases based on whether a wall or a normal box is involved, corresponding to V2 and V3, respectively.



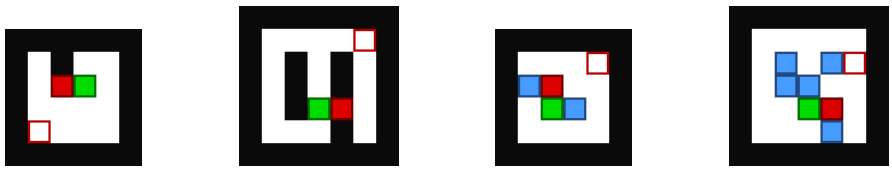
(a) (4 × 4, 7 steps) (b) (5 × 5, 12 steps) (c) (4 × 4, 7 steps) (d) (5 × 5, 8 steps)

Figure 5.6: Examples of V2-OSDE-P2-1 and V3-OSDE-P2-1 problems with their grid size and optimal plan length



(a) (4 × 4, 6 steps) (b) (5 × 5, 11 steps) (c) (4 × 4, 6 steps) (d) (5 × 5, 13 steps)

Figure 5.7: Examples of V2-OSDE-P2-2 and V3-OSDE-P2-2 problems with their grid size and optimal plan length



(a) (4 × 4, 17 steps) (b) (5 × 5, 38 steps) (c) (4 × 4, 7 steps) (d) (5 × 5, 9 steps)

Figure 5.8: Examples of V2-OSDE-P3 and V3-OSDE-P3 problems with their grid size and optimal plan length

Additionally, we created similar tests for pattern three of Section 4.4.3. In these scenarios, the agent is one step away from pushing the goal box against a wall to align it horizontally or vertically with the goal. However, choosing this action leads to a dead-end because there is a wall or normal box located diagonally. These problems are labeled V2-OSDE-P3 and V3-OSDE-P3.

Besides the tests specified above, we also generate general one-step dead-end tests by creating random problems, generating the successors of the initial state, and then checking whether any dead-end heuristic from Section 4.4.3 applies to at least one of them. These sets are referred to as V2-OSDE and V3-OSDE.

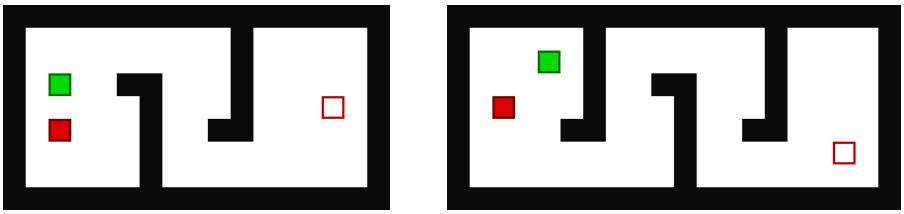
One-Step Dead-End Results

The results are summarized in Table 5.3. We report the number of problems where the agent went into a dead-end, denoted as ”Failed OSDEs” and the normal coverage. Both models rarely fall for the dead-end on the first step, with the exceptions of Model-1 on V2-OSDE-P3 which

fails on 8%, and Model-2-DEH on V2-OSDE-P2-2, which fails on 5%. However, overall coverage for V2-OSDE-P3 is similar for both models, and V2-OSDE-P2-2 remains even significantly higher for Model-2-DEH compared to Model-1. These results demonstrate that both models are capable of detecting and avoiding pattern two and three dead-ends, contradicting our initial assumption that Model-1 would encounter dead-ends more frequently than Model-2-DEH. The results of V2-OSDE and V3-OSDE provide no further insights, as both models rarely fall for the dead-ends in the first step.

	Model-1 (Without Dead-End Heuristic)		**Model-2 (With Dead-End Heuristic)**	
	Failed OSDEs	Coverage	Failed OSDEs	Coverage
V2-OSDE-P2-1	1.3% (1/80)	48.8% (39/80)	1.3% (1/80)	51.3% (41/80)
V2-OSDE-P2-2	0.0% (0/80)	36.3% (29/80)	5.0% (4/80)	68.8% (55/80)
V3-OSDE-P3	8.2% (4/49)	16.3% (8/49)	0.0% (0/49)	12.2% (6/49)
V2-OSDE	0.0% (0/60)	66.7% (40/60)	0.0% (0/60)	56.8% (34/60)
V3-OSDE-P2-1	0.0% (0/80)	56.3% (45/80)	0.0% (0/80)	57.5% (46/80)
V3-OSDE-P2-2	1.3% (1/80)	56.3% (45/80)	0.0% (0/80)	83.8% (67/80)
V3-OSDE-P3	0.0% (0/55)	58.2% (32/55)	0.0% (0/55)	74.0% (37/55)
V3-OSDE	3.3% (2/60)	68.3% (41/60)	0.0% (0/60)	75.0% (45/60)

Table 5.3: Results of the one-step dead-end tests for Model-1 and Model-2-DEH. ”Failed OSDEs” denotes the number of problems where the agent went into a dead-end in the first step, and ”Coverage” represents the normal coverage.



(a) (15 × 7, 34 steps) (b) (18 × 7, 38 steps)

Figure 5.9: Examples of V2-HOOKS problems with their grid size and optimal plan length

5.3.3 Corridor with Hooks

We also created another test set that targets dead-ends, this time specifically for V2. The agent and the goal box start on the left side, while the goal lies on the right. Hooks block the corridor, as illustrated in Figure 5.9, luring the agent to push the goal box towards them, seemingly moving closer to the goal. If the agent takes the bait, it enters a dead-end. The hooks are spaced two cells apart, and the test set is referred to as V2-HOOKS. Note that this pattern is not caught by any heuristic of Section 4.4.3. The objective is to determine if the agent can recognize and avoid these dead-end traps.

The results are shown in Table 5.4. Neither model falls for the dead-end traps, and their behaviors show no significant differences. Beyond having one hook, the solve rates drop swiftly to zero, likely due to too long optimal plan lengths, which exceed those discussed in Section 5.3.1. Generally, neither model navigates beyond the starting region in these cases.

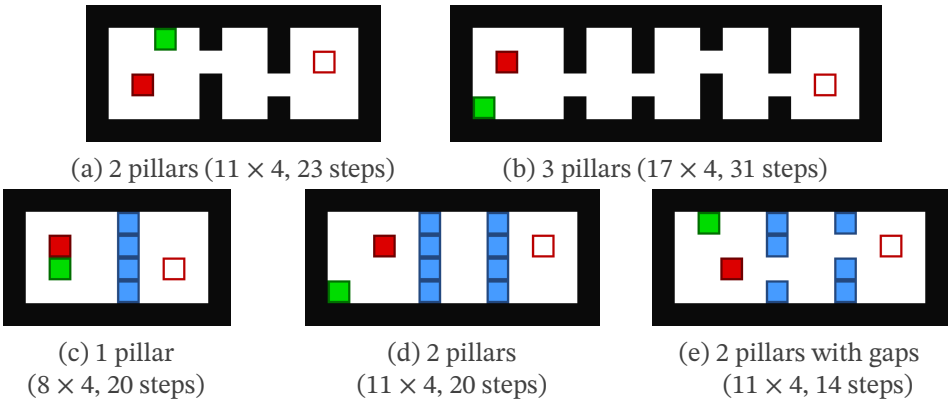
However, it remains unclear if the models understand that they enter dead-ends by pushing the goal box into these hooks. Previously analyzed behavior shows that they are inherently reluctant to push the box against a wall unless the goal is close.

# Hooks:	**Model-1 (Without Dead-End Heuristic)**						**Model-2-DEH (With Dead-End Heuristic)**					
	1		2		3		1		2		3	
	Cat.	Sol.	Cat.	Sol.	Cat.	Sol.	Cat.	Sol.	Cat.	Sol.	Cat.	Sol.
V2-HOOKS	0	3	0	1	–	0	0	3	–	0	–	0

Table 5.4: The results of the corridor with hooks tests for Model-1 and Model-2-DEH, showing the number of problems the agent went into a dead-end-hook denoted as "Cat." (for catched/caught), and the number of problems solved denoted as "Sol.". There are a total of five randomly created problems for each hook count.

5.3.4 Pillars

This test also features a small corridor that forces the agent to navigate through it and aims to determine whether the agent can systematically clear a path. Again, the agent and goal box are initially positioned randomly on the left side, and the goal is on the right. This time, the corridor is blocked by pillars, which are either stacked walls in V2 or normal boxes in V3. In V2, the pillars have a one-cell gap (V2-PILLARS), whereas in V3 pillars either have no gaps (V3-PILLARS) or one-cell gaps (V3-PILLARS-GAP). The pillars are placed with two cells apart. For V3, we also created problems with larger spacing between pillars denoted as V3-PILLARS-SPACE and V3-PILLARS-SPACE-GAP, allowing the agent more free spaces to clear the path. In that case, pillars are placed four cells apart.



(a) 2 pillars (11 × 4, 23 steps) (b) 3 pillars (17 × 4, 31 steps)

(c) 1 pillar (8 × 4, 20 steps) (d) 2 pillars (11 × 4, 20 steps) (e) 2 pillars with gaps (11 × 4, 14 steps)

Figure 5.10: Examples of V2-PILLARS (top row) and V3-PILLARS (bottom row) problems with their grid size and optimal plan length

	Model-1 (Without Dead-End Heuristic)						Model-2-DEH (With Dead-End Heuristic)					
# Pillars:	1	2	3	4	5	6	1	2	3	4	5	6
V2-Pillars	1	0	0	0	0	0	4	2	3	0	1	0
V3-Pillars	5	0	0	0	–	–	5	0	0	0	–	–
V3-Pillars-Gap	4	5	5	3	–	–	5	5	5	4	–	–
V3-Pillars-Space	4	5	5	5	–	–	5	0	0	0	–	–
V3-Pillars-Space-Gap	1	2	5	5	–	–	5	5	5	5	–	–

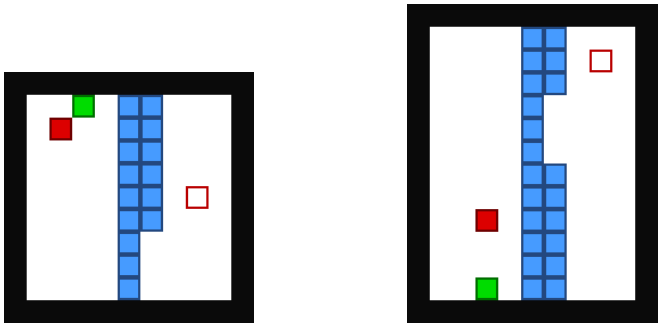
Table 5.5: Results of the pillar tests for Model-1 and Model-2-DEH, showing the absolute number of solved problems by the number of pillars in the problems. There are a total of five randomly created problems for each pillar count.

The results are presented in Table 5.5. For V2, Model-1 either pushes the goal box one cell before the gap of the pillar and then stops or wanders aimlessly, never leaving the starting area. Consequently, it only solves one puzzle in total. In contrast, Model-2-DEH shows better performance, although the plans again do not seem systematic and instead solved by change. For V3-Pillars, both models only manage to solve the problem with one pillar. Model-1 tries to progress to the right but creates dead-ends on its way, whereas Model-2-DEH seems to have no incentive to get closer to the goal. Instead, it pushes the goal box in a circle on the initial position while occasionally pushing a normal box. These patterns apply to all problems with more than one pillar.

When introducing a gap to the pillars of V3, both models perform significantly better, solving nearly all problems. Again, Model-1 makes consistent progress to the right, this time successfully. While Model-2-DEH also resolves most issues, its approach differs from Model-1. It moves the goal box in circles from the left to the right, gradually dismantling pillars when getting close to them. Eventually, the goal box and the agent are close enough to the goal, making the agent's behavior more goal-oriented.

Similar to the gaps, blocking the path to the goal is harder when more space is introduced between the pillars. This again allowed Model-1 to make progress and solve nearly all problems. Model-2-DEH, however, gain failed on problems with more than one pillar showing similar behavior.

The behavior observed in the pillar tests is consistent with the observations made in the increasing puzzles tests of Section 5.3.1. Model-1 has a better understanding of the direction of the goal when it is further away, while Model-2-DEH struggles to identify the correct direction and instead pushes the goal box in random directions or wanders aimlessly.



(a) (9 × 9, 32 steps) (b) (9 × 12, 28 steps)

Figure 5.11: Examples of V2-DOUBLE-PILLAR problems with their grid size and optimal plan length

	Model-1 (Without Dead-End Heuristic)		**Model-2 (With Dead-End Heuristic)**	
	Coverage	PQ	Coverage	PQ
V2-DOUBLE-PILLAR	17.1% (6/35)	2.14	31.4% (11/35)	4.65

Table 5.6: Results of the double pillar tests for Model-1 and Model-2-DEH. ”PQ” stands for the average plan quality.

5.3.5 Double Pillar

Lastly, we created a problem set V2-DOUBLE-PILLAR featuring a single pillar of normal boxes with a width of two. Only a small section of it has a width of one. Again, the agent and goal box are positioned on the left, with the goal on the right. This setup tests the agent’s ability to recognize and navigate this region of width one. Motivation for this set is further outlined in Appendix A.5.

Neither model demonstrates a clear understanding of the task. Model-1 solves 17.1% of the problems, while Model-2-DEH solves 31.4%. The results are summarized in Table 5.6. Analyzing the plan reveals that the models struggle to recognize the narrow passage and fail to approach it effectively, resulting in poor plan quality. When the models solve the problem, the passage is often cleared by chance when the agent gets close to the boxes.

6 Conclusion

This thesis aimed to obtain a better understanding of the challenges of PushWorld in combination with the approach of Ståhlberg *et al.* [28]. To overcome challenges in the original domain and to make the problem more accessible for evaluation, we introduced RestrictedPushWorld. Furthermore, we modified the implementation of Ståhlberg *et al.* [28] to incorporate a replay buffer, as generating complete state space for RestrictedPushWorld is infeasible due to their large sizes. Consequently, we also adapted the validation strategies and integrated other enhancements to the current implementation. We achieved satisfying coverages that exceeded personal expectations, surpassing the original results from Kansky *et al.* [17], even using a smaller fraction of their training problems.

During initial runs, we observed significant differences between the outputs of the actor and critic. The actor consistently displayed 100% confidence in its predictions, while the critic failed to learn, producing nearly equal values for all successor states. As a result, exploration ceased early in training. To mitigate this, we implemented ϵ-policies, added entropy regularization, and split the optimizers to allow different learning rates for the actor and critic heads. Among these, only the latter showed improvements in actor and value function. Nevertheless, the coverage did not increase significantly, and the value function deviates substantially from actual values.

The first observations suggested an issue with dead-ends. To address this, we introduced a dead-end heuristic that aims to maintain high values of the value function during training updates, similar to how Ståhlberg *et al.* [28] handled dead-ends determined from state spaces. While our heuristic is correct, it is incomplete, as determining if a state is a dead-end is equal to determining if the problem is solvable. We provided the idea of a possible proof that this is NP-hard in the case of RestrictedPushWorld. Contrary to initial observations, we could not confirm that training with dead-end heuristics helps to avoid such situations in testing. When confronting the trained models with and without the heuristic with problems specifically designed to trigger dead-ends in the first step, they performed equally well, rarely entering dead-end states. This suggests that the model can learn to identify and avoid the tested dead-end patterns without needing a heuristic to guide it during training. To conclude these findings, one could conduct further experiments with the other dead-end patterns or expand the heuristic to cover more cases in future work.

Besides evaluating dead-ends, we also examined occurrences where the agent wanders aimlessly or pushes the goal box in random directions. This typically occurs when the goal box and goal

are too far apart. We tested the models with and without dead-end heuristics on problems where the distance between the goal box and the goal increases, enabling us to pinpoint when the agent begins to move aimlessly. In these tests, the model without a dead-end heuristic performed significantly better than the other, which showed signs of this behavior even in the problems with the smallest grid size. Future work could explore differences between models from other variants, as our tests were limited to V1.

Other tests focused on the models' ability to systematically find and clear paths blocked by normal boxes. While the models solved some problems, their solutions lacked a recognizable system and were rather found by chance. The model without the dead-end heuristic demonstrated a stronger drive to get the goal box closer to its goal position. In contrast, the model with the dead-end heuristic frequently got stuck in random behavior, as mentioned above. These tests were insufficient to identify clear expressivity issues in the models.

Beyond the approaches discussed here, we also explored adding various derived predicates to the domain, some outlined in Appendix A.5. However, these generally did not improve the models' performance, so we did not include them in the main text. The primary reason for the decreasing performance remains unclear. Additionally, the current implementation of the derived predicates was only temporary and is highly inefficient, iterating over all possible combinations of variables to determine the positive grounded ones. This can slow down the training process immensely.

Despite originally focusing on the expressivity of the GNN, the actor-critic algorithm yielded highly inconsistent results with enormous performance variance, complicating evaluation. Determining whether issues originate from the limited expressivity of the GNN or the training process is challenging. Before proceeding with further evaluations using the current approach, it may be necessary to specify the root cause of the performance variance and develop a more stable training process.

Overall, there are still many open questions and areas for future work. In the short time frame of this thesis, we could not explore all ideas and pinpoint precise reasons for the observed behavior, especially limitations regarding the expressivity of GNNs. However, we believe the results and implementations obtained in this thesis provide a solid foundation for future research.

A Appendix

A.1 PDDL Domains

Listing A.1: PDDL domain of PushWorld with two boxes from Kansky *et al.* [17].

```
(define
  (domain m2)
  (:requirements :strips :typing :conditional-effects
                 :negative-preconditions)

  (:types position - object   moveable-object - object
          direction - object  agent-object - moveable-object)

  (:constants agent - agent-object
              up down left right - direction
              m1 m2 - moveable-object)

  (:predicates
    (should-move ?obj - moveable-object ?dir - direction)
    (has-moved ?obj - moveable-object)
    (at ?obj - moveable-object ?pos - position)
    (connected ?from - position ?to - position ?dir - direction)
    (wall-collision ?obj - moveable-object ?next-pos - position)
    (in-collision
      ?obj - moveable-object
      ?pos - position
      ?other-obj - moveable-object
      ?other-pos - position
    )
  )

  (:action move-agent
    :parameters (?dir - direction)
    :precondition (and
      (not (should-move agent left))
      (not (should-move agent right))
      (not (should-move agent up))
      (not (should-move agent down))
      (not (should-move m1 left))
      (not (should-move m1 right))
      (not (should-move m1 up))
      (not (should-move m1 down))
      (not (should-move m2 left))
      (not (should-move m2 right))
```

```
      (not (should-move m2 up))
      (not (should-move m2 down))
    )
    :effect (and
      (should-move agent ?dir)
      (forall
        (?obj - moveable-object)
        (not (has-moved ?obj)))
    )
  )

  (:action push
    :parameters (
      ?obj - moveable-object
      ?dir - direction ?pos - position
      ?next-pos - position
    )
    :precondition (and
      (should-move ?obj ?dir)
      (not (has-moved ?obj))
      (at ?obj ?pos)
      (connected ?pos ?next-pos ?dir)
      (not (wall-collision ?obj ?next-pos))
    )
    :effect (and
      (not (at ?obj ?pos))
      (at ?obj ?next-pos)
      (has-moved ?obj)
      (not (should-move ?obj ?dir))
      (forall (?other-obj - moveable-object)
        (when
          (and
            (not (has-moved ?other-obj))
            (exists (?other-pos - position)
              (and
                (at ?other-obj ?other-pos)
                (in-collision ?obj ?next-pos ?other-obj ?other-pos)
              )
            )
          )
          (should-move ?other-obj ?dir)
        )
      )
    )
  )
)
```

Listing A.2: PDDL domain of Sokoban from Tarrasch [31].

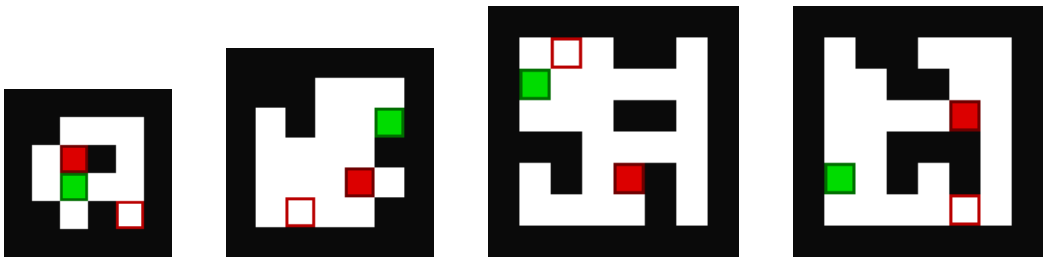
```
(define
  (domain sokoban-domain)
  (:requirements :strips)

  (:predicates (has_player ?x) (has_box ?x)
               (adjacent ?x ?y) (adjacent_2 ?x ?y))

  (:action move-player
    :parameters (?x ?y)
    :precondition (and
      (adjacent ?x ?y)
      (not (has_box ?y))
      (has_player ?x)
    )
    :effect (and
      (has_player ?y)
      (not (has_player ?x))
    )
  )

  (:action push-box
    :parameters (?x ?y ?z)
    :precondition (and
      (has_player ?x)
      (has_box ?y)
      (not (has_box ?z))
      (adjacent ?x ?y)
      (adjacent ?y ?z)
      (adjacent_2 ?x ?z)
    )
    :effect (and
      (has_box ?z)
      (not (has_box ?y))
      (has_player ?y)
      (not (has_player ?x)))
    )
  )
```

Listing A.3: Modified PDDL domain of Sokoban from Tarrasch [31] called RestrictedPushWorld. This version includes boxes without a goal and does not need the `:negative-prconditions` requirement.

```
(define
  (domain sokoban)
  (:requirements :strips)

  (:predicates
    (adjacent ?x ?y)
    (adjacent_2 ?x ?y)
    (has_normal_box ?x) (not_has_normal_box ?x)
    (has_goal_box ?x) (not_has_goal_box ?x)
    (has_player ?x) (not_has_player ?x)
  )

  (:action move-player
    :parameters (?x ?y)
    :precondition (and
      (adjacent ?x ?y)
      (not_has_normal_box ?y)
      (not_has_goal_box ?y)
      (has_player ?x)
    )
    :effect (and
      (has_player ?y)
      (not (not_has_player ?y))
      (not (has_player ?x))
      (not_has_player ?x)
    )
  )

  (:action push-box
    :parameters (?x ?y ?z)
    :precondition (and
      (has_player ?x)
      (has_normal_box ?y)
      (not_has_normal_box ?z)
      (not_has_goal_box ?z)
      (adjacent ?x ?y)
      (adjacent ?y ?z)
      (adjacent_2 ?x ?z)
    )
    :effect (and
      (has_normal_box ?z)
      (not (not_has_normal_box ?z))
      (not (has_normal_box ?y))
      (not_has_normal_box ?y)
```

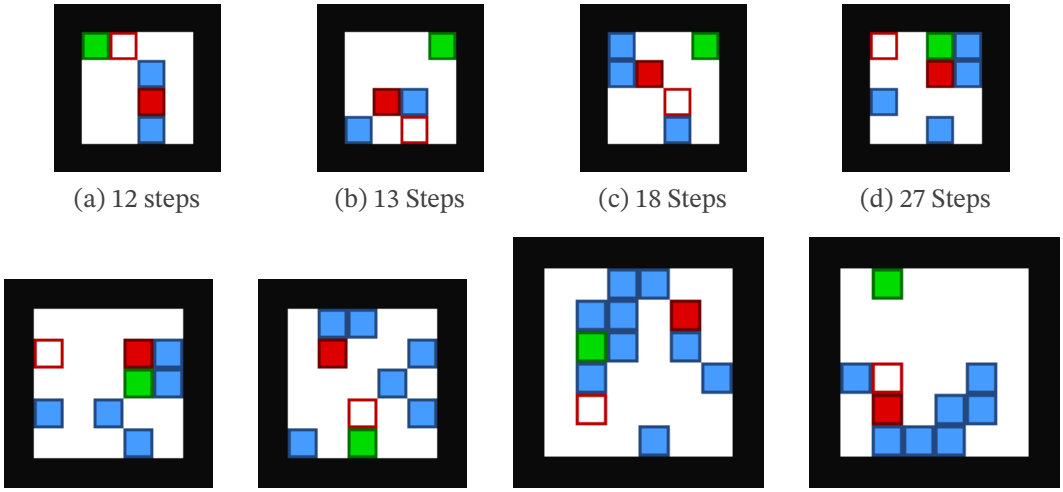
```
45        (has_player ?y)
46        (not (not_has_player ?y))
47        (not (has_player ?x))
48        (not_has_player ?x)
49      )
50    )
51
52    (:action push-box
53      :parameters (?x ?y ?z)
54      :precondition (and
55        (has_player ?x)
56        (has_goal_box ?y)
57        (not_has_goal_box ?z)
58        (not_has_normal_box ?z)
59        (adjacent ?x ?y)
60        (adjacent ?y ?z)
61        (adjacent_2 ?x ?z))
62      :effect (and
63        (has_goal_box ?z)
64        (not (not_has_goal_box ?z))
65        (not (has_goal_box ?y))
66        (not_has_goal_box ?y)
67        (has_player ?y)
68        (not (not_has_player ?y))
69        (not (has_player ?x))
70        (not_has_player ?x)
71      )
72    )
73  )
```

A.2 Collection of Challenging RestrictedPushWorld Puzzles



(a) 18 Steps (b) 23 steps (c) 31 Steps (d) 35 Steps

Figure A.1: Challenging V2 RestrictedPushWorld puzzles with their optimal plan length



(a) 12 steps (b) 13 Steps (c) 18 Steps (d) 27 Steps

(e) 29 Steps (f) 34 Steps (g) 43 Steps (h) 48 Steps

Figure A.2: Challenging V3 RestrictedPushWorld puzzles with their optimal plan length

A.3 Hyperparameters and Problem Sets

Learning Rates		ϵ-Policy		Dead-End	V3 Coverage		
α	β	ϵ	Decay Per Epoch	Heuristic	Closed Determinstic	Closed Stochastic	Open Stochastic
0.0002	0.0002	0	–	No	68%	68%	32%
0.0002	0.0002	0	–	Yes	72%	72%	28%
0.0002	0.0002	0.1	0.8	No	56%	56%	30%
0.0002	0.0002	0.1	0.9	No	52%	52%	27%
0.000 02	0.0002	0	–	No	66%	68%	38%
0.000 02	0.0002	0	–	Yes	55%	53%	46%
0.000 02	0.0002	0.05	0.8	Yes	60%	59%	48%
0.000 015	0.0002	0	–	No	65%	69%	57%
0.000 015	0.0002	0	–	Yes	67%	69%	51%
0.000 01	0.0002	0	–	No	56%	61%	43%
0.000 002	0.0002	0	–	No	62%	59%	48%
0.000 002	0.0002	0.1	0.8	No	57%	49%	60%
0.000 02	0.0004	–	–	No	44%	46%	39%
0.000 004	0.0004	–	–	No	43%	45%	25%
0.0002	0.002	–	–	No	Error: Updates too large.		

Table A.1: The combinations of hyperparameters that we tried and their respective V3 coverage. The default values before modifications were $\alpha = 0.0002$, $\beta = 0.0002$, $\epsilon = 0$ (thus no decay), and no dead-end heuristic.

		Squared Grid Size								
		4	5	6	7	8	9	10	15	Σ
V1	Train	10	10	10	10	10	10	10	0	70
	Test	10	10	10	0	10	0	10	10	60
	Validation	5	0	5	0	5	0	5	0	20
V2, V3	Train	50	50	50	50	50	50	50	0	350
	Test	20	20	20	0	20	0	20	20	120
	Validation	20	0	20	0	20	0	0	0	60
Pw*	Train	–	250	–	–	–	–	–	–	250
	Test	–	200	–	–	–	–	–	–	200
	Validation	–	50	–	–	–	–	–	–	50

Table A.2: An overview of each variant's train, test, and validation set composition. Pw* denotes PwBase, PwLargerPuzzles, PwMoreWalls, and PwMoreObstacles. The test set of these variants is equivalent to the one from Kansky *et al.* [17] converted to RestrictedPushWorld's PDDL. Similarly, the training and validation sets are subsets of their training sets. Problems of V2 and V3 are further separated by densities, which represent the percentage of cells that are walls for V2 or boxes for V3 (rounded down). Their train set consists of densities 10% and 25% with ten problems and densities 15% and 20% with 15 problems for each grid size. The validation and test set consists of densities 10%, 15%, 20%, and 25%, each with five problems for each grid size.

A.4 Grid Size and Density Distributions

		Density of Walls				
	Grid Size	10%	15%	20%	25%	Σ/#
Model-1 (Without Dead-End Heuristic)	4 × 4	5	4	4	5	18/20 (90%)
	5 × 5	5	3	2	4	14/20 (70%)
	6 × 6	3	1	3	3	10/20 (50%)
	8 × 8	4	4	2	3	13/20 (65%)
	10 × 10	3	1	1	0	5/20 (25%)
	15 × 15	4	0	1	0	5/30 (25%)
	Σ/# (%)	24/30 (80%)	13/30 (43%)	13/30 (43%)	15/30 (50%)	65/120 (54%)
Model-2-DEH (With Dead-End Heuristic)	4 × 4	4	4	4	4	16/20 (80%)
	5 × 5	4	3	2	4	13/20 (65%)
	6 × 6	4	1	1	3	9/20 (45%)
	8 × 8	4	3	1	3	11/20 (55%)
	10 × 10	3	1	1	0	5/20 (25%)
	15 × 15	2	0	0	0	2/20 (10%)
	Σ/# (%)	21/30 (70%)	12/30 (40%)	9/30 (30%)	14/30 (47%)	56/120 (47%)

Table A.3: The test coverage distribution of grid sizes and wall densities on V2 for Model-1 and Model-2-DEH. Each combination has five problems.

	Grid Size	Density of Blocks 10%	15%	20%	25%	Σ/#
Model-1 (Without Dead-End Heuristic)	4 × 4	5	3	1	1	10/20 (50%)
	5 × 5	5	4	5	1	15/20 (75%)
	6 × 6	5	4	4	2	15/20 (75%)
	8 × 8	4	4	4	2	14/20 (70%)
	10 × 10	5	5	4	2	16/20 (80%)
	15 × 15	4	4	3	2	13/20 (65%)
	Σ / # (%)	28/30 (93%)	24/30 (80%)	21/30 (70%)	10/30 (33%)	83/120 (69%)
Model-2-DEH (With Dead-End Heuristic)	4 × 4	5	5	3	2	15/20 (75%)
	5 × 5	5	3	4	1	13/20 (65%)
	6 × 6	5	3	3	2	13/20 (65%)
	8 × 8	5	5	3	3	16/20 (80%)
	10 × 10	5	3	4	2	14/20 (70%)
	15 × 15	5	2	2	3	12/20 (60%)
	Σ / # (%)	30/30 (100%)	21/30 (70%)	19/30 (63%)	13/30 (43%)	83/120 (69%)

Table A.4: The test coverage distribution of grid sizes and block densities on V3 for Model-1 and Model-2-DEH. Each combination has five problems.

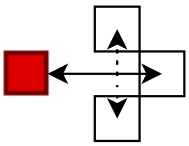
A.5 Adjacent Three

One apparent problem of the RestrictedPushWorld domain is its lack of defined directions, relying solely on adjacencies of distance one and two in single directions. While this efficiently encodes the domain, it causes all problems to become isomorphic to rotated and flipped counterparts.

Another issue that arises is that it complicates pinpointing and referencing the relative position of cells in C_2. For instance, when trying to determine if any cell that is exactly three cells away from the goal block along the same axis (whether three cells to the right, left, up, or down) fulfills a formula ψ using the GC_2 formula:

$$\varphi(x) \doteq \text{has\_goal\_box}(x) \wedge \exists y(\text{adjacent}(x,y) \wedge \exists x(\text{adjacent}(y,x) \wedge \exists y(\text{adjacent}(x,y) \wedge \psi(y))))$$

we cannot distinguish between the three highlighted cells (in any direction):



Instead we require at least three different variables:

$$\varphi(x) \doteq \text{has\_goal\_box}(x) \wedge \exists y \left(\text{adjacent}(x, y) \wedge \exists z \left(\text{adjacent}(y, z) \wedge \text{adjacent\_2}(x, z) \right.\right.$$
$$\left.\left. \wedge \exists x \left(\text{adjacent}(z, x) \wedge \text{adjacent\_2}(y, x) \wedge \psi(x)\right)\right)\right)$$

When referencing greater distances, one would need to introduce even more variables.

We have not yet discovered any explicit problems caused by this in the GNN, and it is unclear what impact it has. Nevertheless, we tried to add derived predicates to the domain, which theoretically could help the GNN overcome this issue, testing if it would improve the model's performance. We tried two approaches:

1. **Explicitly defining directions:** We added directional predicates to the domain: `adjacent_left`, `adjacent_right`, `adjacent_up`, and `adjacent_down`. These would allow us to reference the three cells in the example above by using any directional adjacency three times.
2. **Adding larger adjacencies:** We introduced predicates for adjacencies of distance three or larger: `adjacent_3`, `adjacent_4`, ... This is more restrictive than the first approach, as it only allows reference on single axis.

Both predicates did not improve the model's performance; in fact, they decreased it. A possible reason is that too much information complicates the model's learning process, as the graph becomes more complex and harder to interpret.

Another example worth mentioning is the dead-end pattern five of Section 4.4.3 with the goal and normal boxes:

In GC_2, it is only possible to formulate its existence by exploiting the fact that there is only a single goal box in the problem:

$$\exists x \left(\text{has\_normal\_box}(x) \wedge \exists^{\geq 2} y \left(\text{adjacent}(x, y) \wedge \text{has\_normal\_box}(y) \right.\right.$$
$$\left.\left. \wedge \exists x \left(\text{adjacent}(y, x) \wedge \text{has\_goal\_box}(x)\right)\right)\right)$$

As soon as multiple goal boxes exist, the formula becomes ambiguous and cannot be used to identify the pattern. Also, the square pattern with only normal boxes cannot be identified with the current encoding, which we tried to exploit in Section 5.3.5.

List of Acronyms

AI	Artificial Intelligence
ANN	Artificial Neural Network
CEA	Context-Enhanced Additive Heuristic
CNF	Conjunctive normal form
CNN	Convolutional Neural Network
DP	Dynamic Programming
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
FIFO	First-In-First-Out
GNN	Graph Neural Network
MDP	Markov Decision Process
ML	Machine Learning
MLP	Multilayer Perceptron
PDDL	Planning Domain Definition Language
PI	Policy Iteration
PPO	Proximal Policy Optimization
RGD	Recursive Graph Distance
R-GNN	Relational Graph Neural Network
RL	Reinforcement Learning
STRIPS	Stanford Research Institute Problem Solver
VI	Value Iteration

List of Symbols

General

$\doteq$	equality that is true by definition
$\propto$	proportional to
$\times$	cartesian product of two sets
$\varnothing$	empty set
$\mathbb{N}$	set of natural numbers excluding 0
$\mathbb{R}$, $\mathbb{R}^+$	set of real numbers, set of positive real numbers
$\mathbb{R}^k$	cartesian product of $\mathbb{R}$ k-times ($\mathbb{R} \times \cdots \times \mathbb{R}$)
$\mathbb{E}[X]$	expected value of a random variable X
$\mathbb{P}[X = x]$	probability that a random variable X takes on the value x
$f : X \to Y$	function f that maps elements from a set X to a set Y
$\arg\min_a f(a)$	optimal value of a minimizing the expression $f(a)$
$\arg\max_a f(a)$	optimal value of a maximizing the expression $f(a)$
$\exp x$, $\ln x$	exponential function of x, natural logarithm of x
$\subset$, $\subseteq$	proper subset of, proper subset of or equal to
$\in$, $\notin$	element of, not element of
$\lvert\cdot\rvert$	absolute value of a scalar or number of elements in a set
$\langle\cdot,\cdot,\cdot\rangle$	tuple of multiple variables
$[a, b]$	closed real interval (describes the set $\{x \in \mathbb{R} \lvert a \leq x \leq b\}$)
∇	gradient
$\mathcal{L}$, $\mathcal{L}_{\text{val}}$, $\mathcal{L}_{\text{val}*}$	loss, validation loss, optimal, validation loss
$\widehat{\cdot}$	estimate of a variable, function or expression
$\binom{n}{k}$	binomial coefficient

Logic and Complexity

$\equiv$, $\exists$, $\forall$	logical equivalence, existential quantifier, universal quantifier
FO, FO_k	first-order logic, first-order logic with at most k variables
C, C_k, GC	counting logic, counting logic with at most k variables, guarded counting logic
φ, ϕ, ψ	boolean formulas
V, v_i, C, c_j	set of all variables, i-th variable, set of all clauses, j-th clause
x_i, $\overline{x}_i$	positive literal, negative literal
G_ϕ	graph of a boolean formula ϕ in CNF
$\varkappa$	Hamilton cycle
P, NP	complexity class P, complexity class NP

Classical Planning Model

S	set of all states
S_G	set of all goal states
$A, A(s)$	set of all actions, set of all actions that are applicable in state s
s_0	initial state
$f(a, s)$	transition function specifying the next state when applying action a in state s
$c(a, s)$	cost function specifying the cost of applying action a in state s

STRIPS

F, O	set of all atoms, set of all actions
I, G	set of all initial atoms, set of all goal atoms
$Pre(o)$	set of preconditions for action o
$Add(o)$	set of positive effects for action o
$Del(o)$	set of negative effects for action o

Markov Decision Process

$\mathcal{S}, \mathcal{A}, \mathcal{R}$	set of all states, set of all actions, set of all rewards
$p(s', r \mid s, a)$	transition function specifying a probability of transitioning from state s to state s' with reward r when taking action a
$r(s, a)$	reward function specifying the expected immediate reward when taking action a in state s
$N(s)$	all successors of state s
s, s_t, S, S_t	state, state at time step t, sampled state, sampled state at time step t
a, a_t, A, A_t	action, action at time step t, sampled action, sampled action at time step t
r, r_t, R, R_t	reward, reward at time step t, sampled reward, sampled reward at time step t
s'	successor state of s or just another state independent of s
a'	action after a or just another action independent of a
T	length of an episode
G_t	expected cumulative reward from time step t on
γ	discount factor

Policies and Value Functions

θ, ω	policy parameters, value function parameters
π, π_*, π_θ	policy, optimal policy, policy with parameters θ
$\pi(s)$	deterministic policy specifying the action to take in state s
$\pi(a \mid s)$	stochastic policy specifying the probability of taking action a in state s
$\mathbb{E}_\pi[X = x]$	expected value of a random variable X when following policy π
$v(s), q(s, a)$	state-value/action-value function
$v_\pi(s), q_\pi(s, a)$	state-value/action-value function when following policy π
$v_*(s), q_*(s, a)$	state-value/action-value function when following optimal policy π_*
$v(s, \omega)$	state-value function with parameters ω

α, β	step sizes
$\hat{v}^k(s)$, θ^k, ω^k	estimates during iterative processes each at step k
$b(s)$	baseline with state s
$h(s)$	probability of starting in state s
$\eta_{\pi_\theta}(s)$	expected number of visits to state s under a policy π_θ
$\mu_{\pi_\theta}(s)$	probability of visiting state s under a policy π_θ
$J(\cdot)$	performance measure

Graph Neural Networks

G, V, E	graph, set of all vertices, set of all edges
$N(v)$	set of all neighbors of vertex v
k, L	dimension of feature vectors, number of layers
$f_i(v)$	feature vector of vertex v at layer i
agg, comb	aggregating function, combination function
F	read-out function

List of Figures

2.1 Agent-environment interaction in an MDP, adapted from Sutton and Barto [30] 6
2.2 In C_2 indistinguishable graphs, taken from Horčík and Šír [16] 17
2.3 Euler diagram of the P vs. NP problem, adapted from Esfahbod [6] 17

3.1 Three distinct PushWorld puzzles 21
3.2 Example of a PushWorld puzzle that involves tool use 21
3.3 Three examples of Sokoban puzzles 26

4.1 Overview of the hardness and its reason for each variant 30
4.2 Modified RestrictedPushWorld modules alongside their symbols 32
4.3 Example of a monotone linked planar graph G_ϕ and the corresponding RestrictedPushWorld problem 33
4.4 Assembled goal chain alongside its symbols 33
4.5 Modified RestrictedPushWorld modules from Figure 4.2 without walls 34

5.1 Coverage on V2 by grid size and density of walls 42
5.2 Coverage on V3 by grid size and density of blocks 42
5.4 Examples of V1-INCREASING problems 47
5.5 The number of problems passed on V1-INCREASING 48
5.6 Examples of V2-OSDE-P2-1 and V3-OSDE-P2-1 problems 49
5.7 Examples of V2-OSDE-P2-2 and V3-OSDE-P2-2 problems 49
5.8 Examples of V2-OSDE-P3 and V3-OSDE-P3 problems 49
5.9 Examples of V2-HOOKS problems 50
5.10 Examples of V2-PILLARS and V3-PILLARS problems 51
5.11 Examples of V2-DOUBLE-PILLAR problems 53

A.1 Challenging V2 RestrictedPushWorld puzzles with their optimal plan length . 60
A.2 Challenging V3 RestrictedPushWorld puzzles with their optimal plan length . 61

List of Tables

5.1 Results showing coverage, plan quality, and validation scores 41
5.2 Original results from Kansky *et al.* [17] 41
5.3 Results of the one-step dead-end tests 50
5.4 Results of corridor with hooks tests 51
5.5 Results of pillar tests 52
5.6 Results of the double pillar tests 53

A.1 Hyperparameter combinations with their respective V3 coverage 61
A.2 Each variant's train, test, and validation set composition 62
A.3 Coverage distribution of grid sizes and densities on V2 62
A.4 Coverage distribution of grid sizes and densities on V3 63

List of Listings

A.1 PushWorld's PDDL domain 56
A.2 Sokoban's PDDL domain 58
A.3 RestrictedPushWorld's PDDL domain 59

List of Algorithms

1 Standard Actor-Critic for generalized planning from Ståhlberg *et al.* [28] 19
2 All-Actions Actor-Critic from Ståhlberg *et al.* [28] 19
3 Replay Buffer Fill 35

List of References

[1] C. Aeronautiques, A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. Sri, A. Barrett, D. Christianson, *et al.*, "Pddl - the planning domain definition language," *Technical Report, Tech. Rep.*, 1998.

[2] P. Barceló, E. V. Kostylev, M. Monet, J. Pérez, J. Reutter, and J.-P. Silva, "The logical expressiveness of graph neural networks," in *8th International Conference on Learning Representations (ICLR 2020)*, 2020.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, 2016. eprint: arXiv:1606.01540. [Online]. Available: https://github.com/openai/gym.

[4] S. Cook, "The p versus np problem," *Clay Mathematics Institute*, vol. 2, no. 6, p. 3, 2000.

[5] D. Dor and U. Zwick, "Sokoban and other motion planning problems," *Computational Geometry*, vol. 13, no. 4, pp. 215–228, 1999.

[6] B. Esfahbod, *P np np-complete np-hard.svg*, https://commons.wikimedia.org/wiki/File:P_np_np-complete_np-hard.svg, Accessed: 2024-08-27.

[7] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, no. 3, pp. 189–208, 1971, ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(71)90010-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0004370271900105.

[8] M. Fox and D. Long, "Pddl2.1: An extension to pddl for expressing temporal planning domains," *Journal of artificial intelligence research*, vol. 20, pp. 61–124, 2003.

[9] H. Geffner and B. Bonet, *A Concise Introduction to Models and Methods for Automated Planning: Synthesis Lectures on Artificial Intelligence and Machine Learning*, first edition. Morgan & Claypool Publishers, 2013, ISBN: 1608459691.

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[11] A. Green, B. J. Reji, ChrisE2018, C. Muise, E. Scala, F. Meneguzzi, F. M. Rico, H. Stairs, J. Dolejsi, M. Magnaguagno, and J. Mounty, *Planning.wiki - the ai planning and pddl wiki*, https://planning.wiki/, Accessed: 2024-09-01.

[12] M. Grohe, "The logic of graph neural networks," in *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, IEEE, 2021, pp. 1–17.

[13] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.

[14] M. Helmert and H. Geffner, "Unifying the causal graph and additive heuristics.," in *ICAPS*, 2008, pp. 140–147.

[15] M. W. Hoffman, B. Shahriari, J. Aslanides, G. Barth-Maron, N. Momchev, D. Sinopalnikov, P. Stańczyk, S. Ramos, A. Raichuk, D. Vincent, L. Hussenot, R. Dadashi, G. Dulac-Arnold, M. Orsini, A. Jacq, J. Ferret, N. Vieillard, S. K. S. Ghasemipour, S. Girgin, O. Pietquin, F. Behbahani, T. Norman, A. Abdolmaleki, A. Cassirer, F. Yang, K. Baumli, S. Henderson, A. Friesen, R. Haroun, A. Novikov, S. G. Colmenarejo, S. Cabi, C. Gulcehre, T. L. Paine, S. Srinivasan, A. Cowie, Z. Wang, B. Piot, and N. de Freitas, "Acme: A research framework for distributed reinforcement learning," *arXiv preprint arXiv:2006.00979*, 2020. [Online]. Available: https://arxiv.org/abs/2006.00979.

[16] R. Horčík and G. Šír, "Expressiveness of graph neural networks in planning domains," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 34, no. 1, pp. 281–289, May 2024. DOI: 10.1609/icaps.v34i1.31486. [Online]. Available: https://ojs.aaai.org/index.php/ICAPS/article/view/31486.

[17] K. Kansky, S. Vaidyanath, S. Swingle, X. Lou, M. Lazaro-Gredilla, and D. George, *Push-world: A benchmark for manipulation planning with tools and movable obstacles*, 2023. arXiv: 2301.10289 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2301.10289.

[18] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1412.6980.

[19] D. Lichtenstein, "Planar formulae and their uses," *SIAM journal on computing*, vol. 11, no. 2, pp. 329–343, 1982.

[20] N. Lipovetzky and H. Geffner, "Best-first width search: Exploration and exploitation in classical planning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, 2017.

[21] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, *Asynchronous methods for deep reinforcement learning*, 2016. arXiv: 1602.01783 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1602.01783.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[23] A. Muldal, Y. Doron, J. Aslanides, T. Harley, T. Ward, and S. Liu, *Dm_env: A python interface for reinforcement learning environments*, 2019. [Online]. Available: http://github.com/deepmind/dm_env.

[24] A. Pilz, "Planar 3-sat with a clause/variable cycle," *Discrete Mathematics & Theoretical Computer Science*, vol. 21, no. Discrete Algorithms, 2019.

[25] S. Richter and M. Westphal, "The lama planner: Guiding cost-based anytime planning with landmarks," *Journal of Artificial Intelligence Research*, vol. 39, pp. 127–177, 2010.

[26] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko, "A gentle introduction to graph neural networks," *Distill*, 2021, https://distill.pub/2021/gnn-intro. DOI: 10.23915/distill.00033.

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

[28] S. Ståhlberg, B. Bonet, and H. Geffner, "Learning General Policies with Policy Gradient Methods," in *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning*, Aug. 2023, pp. 647–657. DOI: 10.24963/kr.2023/63. [Online]. Available: https://doi.org/10.24963/kr.2023/63.

[29] S. Ståhlberg, B. Bonet, and H. Geffner, *Learning general policies for classical planning domains: Getting beyond c2*, 2024. arXiv: 2403.11734 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2403.11734.

[30] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, second edition. MIT Press, Nov. 13, 2018, ISBN: 978-0-262-03924-6.

[31] C. Tarrasch, *Sokoban domain pddl file*, https://github.com/Tarrasch/sokoban-planner/blob/master/sokoban-pddls/sokoban-domain.pddl, Accessed: 2024-06-30.