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.

On Limits of GNNs for Planning in Pushworld

Über die Grenzen von Graph-Neuronalen Netzwerken bei Planung in Pushworld

Master Thesis
Masterarbeit

Presented by / Vorgelegt von

Yannik Hesse
406800

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, October 22, 2025

Abstract

Artificial intelligence and reasoning remain vibrant and rapidly evolving research areas. Recent advances in classical planning increasingly leverage deep learning to address complex planning tasks. Reinforcement learning (RL) offers a promising approach for planning, but its effectiveness is often constrained by sparse rewards. Existing training strategies typically rely on supervised learning from optimal classical planners or are limited to small state spaces that allow exhaustive sampling.

In this work, we introduce AV*, a novel method that integrates heuristic search with a learned value function, enabling effective training in highly complex planning domains. Our evaluation focuses on PushWorld, a challenging benchmark that tests tool use and reasoning over intricate object shapes in long-horizon tasks. To overcome limitations of prior approaches, we extend relational graph neural networks (R-GNNs) with global attention pooling and incorporate PushWorld-specific logical propositions, enhancing the model's expressive power.

Experimental results show that AV* outperforms pure policy-gradient RL in complex domains. In PushWorld, R-GNNs generalize better than conventional CNN-based architectures. Combined with search-based rollouts, AV* achieves performance competitive with the robust classical planner LAMA, demonstrating strong domain-specific reasoning. These results suggest that integrating learned value functions, heuristic search, and structured logical representations offers a promising path toward scalable and generalizable planning solutions.

Contents

1 Introduction

Reasoning, and with it planning, has become increasingly central to recent advances in Artificial Intelligence (AI) research. We expect AI models, particularly modern language models, to reason effectively and generate plans that generalize across domains. Yet, even the most advanced planning systems still struggle to reach human-level performance in relatively simple tasks. If a model cannot reliably plan how to make a perfect breakfast, how can we expect it to navigate the complexities of real-world, multi-agent scenarios such as autonomous driving? To evaluate planning and reasoning models, we often rely on much simpler domains, which we understand and can interpret, so that we can scale them to generate arbitrarily difficult tasks.

One such reasoning domain is *PushWorld*, introduced by DeepMind in 2023 [25]. *PushWorld* is a challenging grid-based planning and reasoning task in which an agent must push blocks in a 2D environment. The complexity arises from irregularly shaped objects, tool use, and the need for long-horizon planning over multiple time steps. In many ways, *PushWorld* is a "needle in a haystack" problem, where a single wrong move can render the task unsolvable.

While there has been limited work on applying AI reasoning models to *PushWorld*, significant attention has been given to similar combinatorial domains, most notably *Sokoban*. Numerous approaches have attempted to solve *Sokoban* using advanced search algorithms, heuristics, and data generation techniques. A key difference between the two domains lies in their focus. *PushWorld* emphasizes generalizable planning. It can be seen as a foundational toolkit for constructing complex, puzzle-like environments. These environments feature diverse interaction effects, arising from multi-object pushing and objects of arbitrary shapes.

In contrast to *Sokoban*, whose difficulty stems from combinatorial complexity, *PushWorld* presents a very different challenge. In *Sokoban*, there is often only a single path to the goal, and a single wrong move can cause failure. Hard instances are nearly impossible for humans, and AI systems have long surpassed human performance. In *PushWorld*, humans solve the problems surprisingly well. Yet even state-of-the-art classical planners struggle to find solutions. This discrepancy likely arises from the exponentially large state space and the lack of guidance about whether a given path leads toward the solution. It highlights a fundamental limitation: current AI systems lack the intuitive reasoning humans use to efficiently navigate complex problem spaces.

Efforts to bridge this gap have explored various strategies, though many rely on handcrafted, problem-specific, knowledge, which is costly and non-scalable. An example is the "Novelty+RGD" approach introduced by Kansky et al. [25]. More recently, deep learning has

emerged as a powerful tool in addressing such problems. High-profile successes such as AlphaGo [34], OpenAI Five [5], and DreamerV3 [19] demonstrate that combining learning and planning can solve previously intractable tasks. These systems suggest that learned domain knowledge can support high-level reasoning and long-term planning.

In the context of planning, the idea of using specialized models equipped with domain knowledge is not new. However, systematically learning this knowledge in a generalizable way remains a major challenge. One key difficulty lies in representing planning problems and how models perceive domain instances. Recent work by Ståhlberg et al. [36] proposes encoding logical propositions of planning problems using GNNs and learning general policies using Reinforcement Learning (RL). Their approach integrates structural invariances, such as grid rotation and mirroring, which supports generalization across problem instances. This allows the same model to reason over variable grid sizes and object numbers, something that more traditionally used representations, such as CNNs [27] struggle with.

However, GNNs are not without limitations. They suffer from phenomena such as *oversmoothing* [28] and *oversquashing* [43] which can degrade performance if not carefully addressed in model design. Moreover, the theoretical expressiveness of GNNs is inherently constrained, unlike Multilayer Perceptrons (MLPs), which are universal function approximators under sufficient scale. Barceló et al. [4] have shown that their logical expressive power can be characterized in terms of two-variable counting logic ($\mathcal{C}_2$) and its guarded variant ($\mathcal{GC}_2$), which are subsets of first-order logic. This introduces an important tension: while GNNs provide a natural and intuitive framework for representing structured domains, they may lack the theoretical capacity to capture the complexity of problems like *PushWorld*. Therefore, careful attention must be given to the match between problem representation and model capacity.

In this thesis, we focus on applying GNNs to solve the *PushWorld* planning problem. Our approach begins by identifying the shortcomings of existing methods and proposing strategies to address them. Furthermore, we investigate the limitations of GNNs in terms of expressive power, examining cases where and how additional information can be provided to overcome limitations.

A key aspect of this work is evaluating how our GNN models generalize compared to the more traditionally used architectures such as CNNs. Generalization, the transfer of knowledge learned from simpler problems to more complex ones, is a critical challenge for traditional learning-based methods, which often struggle in this area. Yet it is an essential capability for planning and reasoning agents that must handle novel scenarios or larger problem instances.

The remainder of this thesis is structured as follows. Chapter 2 introduces the necessary background, including theoretical foundations and core concepts. Chapter 3 reviews related work, focusing on both *PushWorld* and current learning-based planning approaches. Chapter 4 presents our methodology, detailing a scalable training algorithm, identifying the limitations

of current models, and outlining our strategies to address them. Additionally we describes the logical encodings of the domains under consideration. Chapter 5 reports our experimental results, beginning with a simplified hybrid setting between *Sokoban* and *PushWorld*, and then on the full *PushWorld* benchmark, comparing our contributions both against other deep learning methods and against classical planners. Finally, Chapter 6 concludes with a summary of our contributions and an outlook on future research directions.

2 Background

In this chapter, we introduce the most important concepts required to understand our approach in later sections. We begin by presenting the general *PushWorld* problem, providing motivation and highlighting its challenging aspects. Next, we explain the concepts of planning and RL, and discuss how these concepts relate to GNNs. Finally, we explain the concept of logical expressive power of GNNs and examine how it relates to Relational Graph Neural Networks (R-GNNs) in the context of *PushWorld*.

2.1 PushWorld

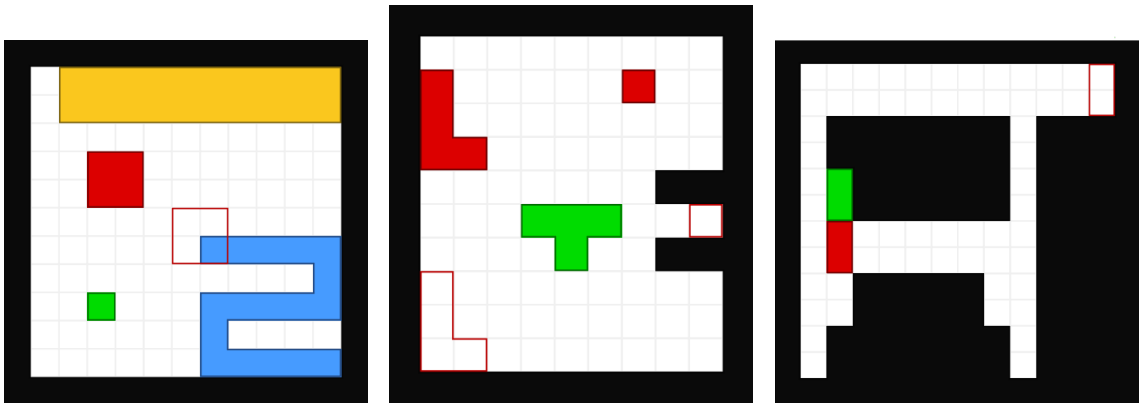
PushWorld [25][1] is a challenging grid-world environment designed to evaluate AI planning and reasoning capabilities in scenarios that require physical manipulation of objects and tools. It presents a suite of handcrafted puzzles demanding complex skills, including path planning, object manipulation, obstacle avoidance, tool utilization, and multi-goal prioritization. This distinguishes it from simpler environments such as *Sokoban*, which involve fewer object types and less dynamic interactions.

The main *PushWorld* domain consists of an agent that can move up, down, left, or right. If this agent (sometimes also called player) moves into another object, it pushes that object one tile in the same direction. This push can have cascading effects: if the pushed object collides with another, if possible all affected objects are moved accordingly in the same direction. The goal is to place the red goal objects, which may have arbitrary shapes, on top of the outlined red goal areas. The benchmark suite, provided by Kansky et al. [25], is divided into five levels, from `level0` (the simplest) to `level4` (the most difficult). While `level0` problems are procedurally generated and intended as a training set for deep learning methods, levels 1–4 contain handcrafted puzzles. Each level reflects a difficulty based on the time humans typically spend solving them, with `level4` being the hardest. An illustration of three *PushWorld* problems can be found in Figure 2.1.

The primary challenges in *PushWorld* arise from the intricate interplay of these differently shaped objects. Here, since objects can not be pulled, actions can have irreversible consequences (a “needle in the haystack” type of problem), requiring pre-emptive strategic positioning. Evaluations have shown that both classical planning algorithms and model-free deep reinforcement learning methods struggle to solve all *PushWorld* puzzles. Classical planners achieve some success with heuristic approaches such as Novelty+RGD [25], but still exhibit limitations in

[1] `https://deepmind-pushworld.github.io/play/`

solving the full range of puzzles within reasonable time constraints. Model-free reinforcement learning methods, specifically Deep Q-Learning (Deep Q-Network) (DQN) [30] and Proximal Policy Optimization (PPO) [35], face even greater difficulties, particularly with the sparse reward structure and multi-goal scenarios. In contrast, humans solve most of these puzzles with relative ease, highlighting a significant gap between current AI performance and human-level problem solving skills in this domain. Humans find these problems easier because we can decompose them into simpler subproblems and reason about how complexly shaped objects must interact to achieve a solution. Even state-of-the-art visual large language models struggle to construct coherent plans, as we have observed empirically on several puzzles from the *level1* problem suite using GPT-4[2].



(a) Problem 1: Two Obstacles (b) Problem 2: A Perfect Fit (c) Problem 3: A Tight Squeeze

Figure 2.1: Examples of the first three *PushWorld* problems. Each puzzle requires the red outlines to be covered by red blocks. The agent may move the green block (as a whole unit) up, down, left, or right. Challenges include walls (black), other movable objects (blue shapes), and restricted areas (yellow), which the agent (green) may not enter but into which the blue and red objects may be pushed.

What makes this benchmark particularly compelling is the challenge it presents: solving it demands a highly general form of intelligence, capable of reasoning about newly encountered complex shapes in much larger grids than previously seen. The building blocks of these problems can take arbitrary shapes (including the agent itself) allowing for a wide variety of puzzles to be generated. Moreover, the grid size can vary significantly, sometimes exceeding 25×25. It is still uncertain to what extent these problems can be solved solely through logical reasoning, though the overall complexity of the domain is likely PSPACE-complete. This is because *Sokoban* is PSPACE-complete [9, 10, 21], and if the *PushWorld* problem is restricted to only include goal-boxes of size 1×1, it closely resembles *Sokoban*, albeit with the added complexity of chained pushing. These observations suggest that *PushWorld* is a very challenging problem and, in the general case, intractable for R-GNNs (introduced later).

[2] `https://chatgpt.com`

2.2 Classical Planning

To lay the groundwork for our approach, we first provide a formal overview of classical planning, showing how planning problems can be viewed as search problems in a state space. We then introduce heuristic search techniques and planning solvers, including the classical planner LAMA and the *A** search algorithm, which are widely used to efficiently find solutions in complex planning domains.

2.2.1 Planning Definition

In classical planning [15], we are given a problem $P = \langle D, I \rangle$, which consists of a domain D and an instance I. The domain provides the general structure and rules of the problem, describing the possible actions and the existing predicates. More formally, a domain D consists of a set of predicates *Preds*, where each predicate has an arity defining the number of objects it can be defined by. A classical example is the problem of building a tower of blocks in *Blocksworld*. Here, a binary predicate could be $on(x, y)$, representing that block x is sitting on top of block y, while a unary predicate could be $red(x)$, indicating that block x is red. If x, y are replaced by objects like $red(block_1)$ we would call this a grounded atom. A set of atoms defines a state.

Additionally, D includes a set of actions that can be applied to a state. In general, an action consists of preconditions and effects. If the preconditions are met, we can apply an action a to a state s_1, denoted as $f(a, s_1) = s_2$, resulting in a new state s_2. For example, an action $change\_color(o)$ might have the precondition $red(o)$ and the effect $\neg red(o) \wedge blue(o)$, meaning the object's color changes from red to blue.

An instance $I = (\text{Objs}, init, goal)$ consists of an immutable set of objects (Objs), an initial state ($init$), and a goal set of (possible negated) atoms ($goal$). A state s_g is considered a goal state if $goal \subseteq s_g$ holds. The objective of planning is to find a sequence of actions (also called a plan) $(a_0, a_1, \dots, a_n)$ such that, when applied to $init$, it results in a goal state s_g, i.e.,

$$f(a_n, \dots, f(a_1, f(a_0, init))) = s_g.$$

In general planning [38], we seek an algorithm that can solve any problem P for a given domain D, without considering the length of the solution plan. In optimal planning, we additionally assign costs to actions and seek a plan that minimizes the cumulative cost.

For our purposes, it is more convenient to work with the notion of a state model $S(P) = \langle S, s_0, S_g, Act, A, f \rangle$, where:

- S is the set of all states reachable from the initial state s_0,
- S_g is the set of all reachable goal states,
- Act is the set of all possible actions,

- $A(s) \subseteq Act$ represents the subset of actions whose preconditions are satisfied in state s
- f is the transition function, where for any action $a \in A(s_i)$, the resulting state is given by

$$s_{i+1} = f(a, s_i).$$

In practice, a common way to work with such planning problems is to use STRIPS [14] and the PDDL language [1]. We can define an action in PDDL as follows

```
(:action change-color
    :parameters (?x)
    :precondition (red(?x))
    :effect (and (not (red ?x)) (blue ?x))
)
```

This has the same effect as the action we described earlier, which changes the color of a red(x) into a blue(x).

2.2.2 Planning Solvers and Search

Solving classical planning problems often relies on search algorithms, specifically heuristic search, which helps prune the search space and guide the algorithm toward the goal efficiently. Planning problems can involve an exponentially number of states, making exhaustive search infeasible. Without effective and optimized pruning strategies, planning solvers would struggle to handle complex problems.

One optimized solver is *LAMA* [32], which won the *International Planning Competition (IPC) 2008*[3]. At its core, LAMA is based on the A* search algorithm [20], which we will also employ in our method later. We provide a sketch of the A* algorithm in Algorithm 1.

A* is an informed search algorithm that utilizes a heuristic function $h(s)$ to estimate the cost from the current state to the goal. This heuristic does not need to be precise or even entirely accurate; even a rough approximation can significantly improve search efficiency by prioritizing more promising paths. For each state its priority is given by the cost it took to reach that state $g(s)$ and the estimated cost to get to the goal $h(s)$. States are then expanded based on this priority $f(s) = g(s) + h(s)$. In general if the state-space is finite and a goal exists A* is guaranteed to terminate and find the goal. If additionally $h(s)$ is admissible it is guaranteed to find the shortest path to the nearest goal state. As such A* ensures an efficient search process, making it one of the most widely used algorithms in planning and AI search problems.

[3] https://www.icaps-conference.org/competitions/

Algorithm 1 A* Search Algorithm

Require: State model $S(P) = \langle S, s_0, S_g, Act, A, f \rangle$, heuristic function $h : S \to \mathbb{R}$
Ensure: A sequence of actions leading from s_0 to a goal state in S_g, if one exists

```
Initialize the open set: O ← {(s_0, ∅)}                    Set of states to explore, storing (state, path)
Initialize the cost map: g(s_0) ← 0                        Cost from start to state
Initialize the priority queue with s_0, priority g(s_0) + h(s_0)
while O is not empty do
    Extract (s, π) from O with the lowest g(s) + h(s)
    if s ∈ S_g then
        return π                                           Return the sequence of actions leading to the goal
    for all actions a ∈ A(s) do
        s' ← f(a, s)                                       Compute successor state
        g' ← g(s) + cost(a)
        if s' is not in O or g' < g(s') then
            Update g(s') ← g'
            Update priority queue with s', priority g(s') + h(s')
            Store π' = π ∪ {a} in O
return failure                                             No solution found
```

2.3 Reinforcement Learning

RL [41] provides a framework for agents to learn decision-making strategies through interaction with an environment. The central aim is to optimize a policy that maximizes the cumulative reward over time. We aim to give a short formal overview of reinforcement learning and also explain a commonly used method for learning policies.

2.3.1 Overview of Reinforcement Learning

RL is most commonly described within the formalism of a Markov decision process (MDP), defined as

$$\text{MDP} = (S, A, P, R, \gamma),$$

where:

- S is a finite set of states,
- A is a finite set of actions available to the agent,
- $P : S \times A \times S \to [0, 1]$ is the state transition probability function. For every state-action pair (s, a),

$$\sum_{s' \in S} P(s, a, s') = 1, \quad \forall\, s \in S,\ a \in A,$$

  where $P(s, a, s')$ denotes the probability of transitioning to state s' after taking action a in state s,

- $R : S \times A \times \mathbb{R} \to [0,1]$ is the reward distribution function, where

$$R(s,a,r) = \Pr(R_t = r \mid S_t = s, A_t = a),$$

  i.e., the probability of receiving reward $r \in \mathbb{R}$ when action a is taken in state s,
- $\gamma \in [0,1]$ is the discount factor, which determines how future rewards are weighted relative to immediate rewards.

Often a direct transition-reward formulation is used $p(s', r|s, a) = \Pr\big(R_t = r, S^{t+1} = s' \mid S_t = s, A_t = a\big)$.

A *policy* defines the agent's behavior in an MDP. Formally, it is a mapping $\pi : S \times A \to [0,1]$, such that for each state $s \in S$,

$$\sum_{a \in A} \pi(s,a) = 1.$$

Here, $\pi(s,a)$ denotes the probability of choosing action a when the agent is in state s. If π is *deterministic*, we typically write $\pi(s) = a$ to indicate that action a is selected with probability one in state s. If π is *stochastic*, an action is sampled at execution time according to the distribution:

$$a \sim \pi(\cdot \mid s).$$

Figure 2.2 illustrates the agent-environment interaction loop in an MDP. At each time step, the agent observes the current state s^t, selects an action a^t according to its policy π, and the environment responds by transitioning to a new state s^{t+1} and providing a reward r^{t+1}. An episode is defined as a sequence of state-action pairs that begins with an initial state s_0 and terminates when a terminal state is reached:

$$\tau(s_0) = \{(s_0, a_0), (s_1, a_1), \dots, (s_n, a_n)\}.$$

The return (i.e., total discounted reward) starting from s_0 is given by

$$G(s_0) = \sum_{t=0}^{n-1} \gamma^t R(s_t, a_t, s_{t+1}).$$

Under a deterministic policy π, the resulting episode can be denoted as

$$\tau_\pi(s_0) = \{(s_0, \pi(s_0)), (s_1, \pi(s_1)), \dots, (s_n, \pi(s_n))\}, \quad \text{with } s_{t+1} \sim P(s_t, \pi(s_t)).$$

Consequently, the return under policy π is

$$G_\pi(s_0) = \sum_{t=0}^{n-1} \gamma^t R(s_t, \pi(s_t), s_{t+1}).$$

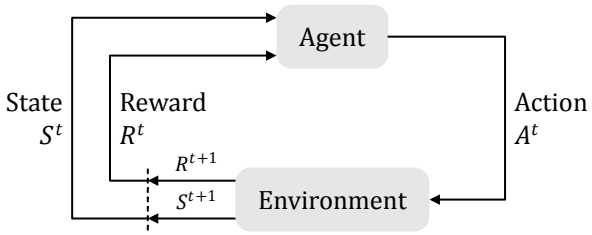


Figure 2.2: Agent-environment interaction in an MDP, adapted from Sutton, Barto, et al. [41]. At each time step, the agent observes the state s^t and reward r^t, selects an action a^t, and transitions to a new state s^{t+1} while receiving a reward r^{t+1}.

The objective in RL is to find an optimal policy π^* that maximizes the expected return:

$$\pi^* = \arg\max_{\pi} \mathbb{E}[G_\pi(s_0)],$$

where the expectation is taken over the stochastic transitions of the environment and, the possibly stochastic, policy.

There are two principal learning-based approaches to solving RL problems:

1. **Value-based Methods:** These methods focus on learning an approximated *value function*. A *value function* indicates how good or bad a specific state is. Given a policy π_1, the value of a state can be estimated as

   $$v_{\pi_1}(s) = \mathbb{E}\big[G_{\pi_1}(s)\big],$$

   representing the expected return when following policy π_1 from state s. The policy π_1 could be a random policy or a learned one. The optimal value function is given by

   $$v_{\pi^*}(s) = v^*(s) = \mathbb{E}[G_{\pi^*}(s)].$$

   A policy can then be derived from a value function by selecting actions that maximize the expected return:

   $$\pi(s) = \arg\max_{a \in A} \sum_{s' \in S} P(s, a, s')\, v(s'),$$

   which is equivalent to using the *action-value function* (or Q-function)

   $$Q(s, a) = \sum_{s' \in S} P(s, a, s')\, v(s'),$$

   so that

   $$\pi(s) = \arg\max_{a \in A} Q(s, a).$$

Since the true transition dynamics P and the optimal value function v^* are generally unknown, these value functions are usually approximated iteratively. The approximated functions are denoted by $\tilde{V}(s)$ and $\tilde{Q}(s,a)$.

2. **Policy-based Methods:** These methods directly parameterize the policy as a probability distribution $\pi(s,a)$ over actions, satisfying

$$\sum_{a \in A} \pi(s,a) = 1.$$

This direct formulation naturally allows for stochastic policies. Although policy-based methods offer several advantages, in this thesis we primarily employ value-based approaches due to their relative simplicity and stability during training. Parameterized policies can be prone to instability or catastrophic forgetting, which is why state-of-the-art poliy-gradient algorithms such as PPO [35] employ additional mechanisms to stabilize learning.

2.3.2 Value Iteration

One common iterative approach to learning a value function is *value iteration*. This method is based on the Bellman optimality equation [41], which states

$$v^*(s) = \max_{a \in A} \sum_{s',r} p(s', r \mid s, a)[r + \gamma\, v^*(s')].$$

This equation expresses the optimal value function in a self-referential manner: the value of a state depends entirely on the values of its successor states and immediate next reward. This property allows us to reformulate the equation into an iterative update rule.

Using this idea, we define the following update for the value function:

$$v^{k+1}(s) = \max_{a \in A} \sum_{s',r} p(s', r \mid s, a)[r + \gamma\, v^k(s')].$$

Here, the value of a state is updated based on the values of its successor states, allowing rewards to gradually propagate throughout the state space. By performing this update for all states repeatedly, it can be shown that, as $k \to \infty$, the value function converges to the optimal value function v^* [41].

This approach, however, requires iterating over all states and that the environment dynamics p are fully known. A practical method to overcome these limitations is *Deep Approximated Value Iteration (DAVI)* [2], which replaces the true value function with a parameterized approximation $\tilde{V}(s)$:

$$\tilde{V}(s) = \max_{a \in A} \sum_{s',r} p(s', r \mid s, a)[r + \gamma\, \tilde{V}(s')].$$

Here, $\tilde{V}$ is a function that can be iteratively learned and improved. While this loses the theoretical guarantees of optimality, in practice such methods have been shown to converge [2] provided that appropriate states s are selected for updating.

2.3.3 Signals in Reinforcement Learning

In value iteration, the agent improves its policy by updating its current estimated value function based on the next states. A meaningful update occurs when the value-iteration error is nonzero:

$$0 \neq \left[\tilde{V}(s) - \max_{a \in A} \sum_{s',r} p(s', r \mid s, a)[r + \gamma \tilde{V}(s')]\right].$$

If this error is exactly zero, it indicates that, according to the agent's current knowledge, the value function has converged to the optimal value function and no further refinement is needed. Conversely, if the error is nonzero, $\tilde{V}$ must adjust its parameters to minimize this discrepancy, thereby improving the policy. The magnitude of this error depends on the successor states s' and is influenced either by immediate rewards r or by delayed rewards implicitly encoded in $\tilde{V}(s')$.

Effective learning requires identifying states s that provide the most informative updates. These transitions, which drive learning progress, are referred to as *learning signals*. Collecting meaningful learning signals can be challenging, especially in environments with sparse rewards. Sparse rewards occur when most state transitions provide the same constant reward (usually 0 or some constant penalty), offering little guidance to the agent.

For example, consider a grid world where the goal is to reach a specific location. If the environment only provides a nonzero reward upon reaching the goal, the agent cannot learn meaningful information about the environment unless it reaches the goal. In such cases, learning signals are rare, creating a sparse reward setting.

2.3.4 Learning Methods in Classical Planning

There have been numerous efforts to integrate reinforcement learning (RL) with classical planning [16, 38]. Planning provides a structured framework for problem-solving, while RL offers a mechanism to optimize decision-making over time. If a planning problem can be formulated as a Markov Decision Process (MDP), RL techniques can be applied, allowing neural networks (discussed in the next section) to replace classical search-based solvers.

This approach offers several potential advantages over traditional search methods. First, RL enables the automated acquisition of domain-specific knowledge, allowing an agent to generalize across multiple problems within the same domain without requiring exhaustive search. Second, classical search methods often struggle in large state spaces without a well-

designed heuristic, whereas learning-based methods such as RL have the potential to scale by learning generalizable heuristics through experience.

However, RL-based planning also introduces challenges. One primary difficulty is the absence of rewards. If a reward is only provided upon finding a valid solution, this creates a sparse reward environment that makes acquiring meaningful learning signals difficult. One way to address this is by engineering problem-specific rewards or leveraging heuristics from classical planners [16].

Compared to the full RL formulation, a simplifying factor in planning is that state transitions are usually deterministic and known, i.e.,

$$P(s, a, s') = 1 \text{ for the successor state } s',$$

which ensures that actions have predictable effects, simplifying the application of RL techniques.

The remaining challenge lies in defining a suitable representation for $\tilde{V}(s)$ that enables effective learning.

2.4 Neural Networks

Neural networks form the foundation of deep learning. At their core, they consist of simple parameterized functions that, when combined, act as universal function approximators [24]. This means that, given suitable parameters and width / depth, a neural network can approximate any continues function. The primary challenge lies in determining these parameters, which is typically accomplished using gradient descent [26]. This optimization process requires these functions that make up the neural networks to be differentiable.

2.4.1 Multi-Layer Perceptron

One of the most fundamental types of neural networks is the Multi-Layer Perceptron (MLP). An MLP consists of layers of interconnected neurons, where each layer applies a linear transformation followed by a non-linear activation function. Mathematically, an MLP layer can be expressed as

$$f(D) = \sigma\big(W^T D\big),$$

where $W \in \mathbb{R}^{n \times m}$ is a learnable weight matrix, $D \in \mathbb{R}^n$ represents the input data, and σ is a non-linear activation function, such as ReLU or sigmoid, which introduces non-linearity into the model.

An MLP is commonly represented as a composition of multiple layers, e.g.,

$$MLP_3(D) = f_3 \circ f_2 \circ f_1(D).$$

By stacking multiple such layers, an MLP can learn complex functions, making it a powerful tool for various machine learning tasks.

However, in many applications, data does not naturally reside in $\mathbb{R}^n$. In such cases, the data must either be encoded into a suitable vector representation or processed using specialized model architectures. For example, Convolutional Neural Networks (CNNs) [27] are commonly used for image data, as their convolutional structure makes them equivariant to translations and robust to local positional shifts. Transformers [44] are widely used for text data because their attention mechanism can flexibly model dependencies in variable-length sequences. Graph Neural Networks (GNNs) [33], on the other hand, are particularly suited for relational data, as they are permutation invariant (for graph-level tasks) or equivariant (for node-level tasks) and can handle inputs of varying sizes.

2.4.2 Graph Neural Networks

Relational data cannot be naturally represented in $\mathbb{R}^n$; thus, standard deep learning models require adaptations to handle such inputs. A natural way to represent relational structures is as a graph $G = (V, E)$, consisting of a set of vertices V and edges $E \subseteq V \times V$. Graph Neural Networks (GNNs) provide a framework to process such data.

One of the most widely used classes of GNNs is message-passing GNNs. In these networks, each node $v \in V$ maintains an embedding, and information is propagated through the graph along edges E via message passing. Depending on the type of graph, node embeddings can include initial features; for example, in a network of cities, these features might represent population size or combined household income.

Message passing operates as follows: at each step, the embedding of a node is updated based on its current state and an aggregation of the embeddings of its neighbors. Formally, this can be expressed as

$$x_v^{i+1} = \mathrm{comb}\big(x_v^i, \mathrm{agg}(x_j^i \mid j \in N_G(v))\big),$$

where $N_G(v) = \{j \mid (j, v) \in E\}$ denotes the set of neighbors of node v, $\mathrm{agg}(\cdot)$ is a permutation-invariant aggregation function such as sum, mean, or max, and $\mathrm{comb}(\cdot)$ is a combination function, which does not need to be permutation-invariant (e.g., a multi-layer perceptron applied to the concatenation of the node embedding and aggregated neighbor features). Permutation invariance in the aggregation function is crucial since the ordering of graph neighbors is arbitrary. This ensures that two isomorphic graphs produce the same output and that the representation does not depend on the order in which relational data is provided.

After L rounds of message passing, each node obtains a final embedding x_v^L. Intuitively, these steps propagate information through the graph, allowing each node to capture increasingly broader relational context.

For tasks such as graph-level regression, node embeddings must be aggregated into a single graph-level representation. This is achieved using a global pooling (aggregation) function

$$E_G = \mathrm{agg}\big(x_v^L \mid v \in V\big), \quad E_G \in \mathbb{R}^n.$$

The resulting graph embedding E_G can then be passed to a readout function, such as a multi-layer perceptron (MLP), to produce a scalar output

$$f(G) = \mathrm{MLP}(E_G) = \mathrm{MLP}\big(\mathrm{agg}\big(x_v^L \mid v \in V\big)\big).$$

A common choice for aggregation is the LogSumExp (LSE) function

$$\mathrm{LSE}(x_1, \dots, x_n) = \log\Big(\sum_{i=1}^{n} \exp(x_i)\Big),$$

which provides a smooth and differentiable approximation of the maximum while maintaining numerical stability.

Despite their flexibility, GNNs have limitations. Information can only propagate up to L hops; thus, dependencies beyond this distance may not be captured. For instance, when using a GNN to determine the shortest path in a graph, L must be large enough to encompass the longest relevant path.

Nonetheless, GNNs offer significant advantages. Their inherent permutation invariance facilitates generalization, and their extensibility to varying numbers of nodes and edges enables them to handle large-scale relational datasets. This makes GNNs particularly well-suited for encoding logical structures in graphs, as discussed in the subsequent sections.

2.4.3 Relational Graph Neural Network

In this thesis, we build upon the relational graph neural network (R-GNN) introduced by Ståhlberg et al. [36]. Their method proposes a PDDL-to-Graph encoding, which provides the final component necessary to enable reinforcement learning for planning problems. In this formulation, the planning problem is represented as a hypergraph in which objects serve as nodes and predicates define the relations between them. To incorporate goal information, each predicate is further extended with a goal-specific variant.

The message-passing rules in an R-GNN differ from those of standard GNNs described in Section 2.4.2. We therefore outline them here. Let $o \in \mathit{Obj}$ denote objects and $p \in \mathit{Preds}$

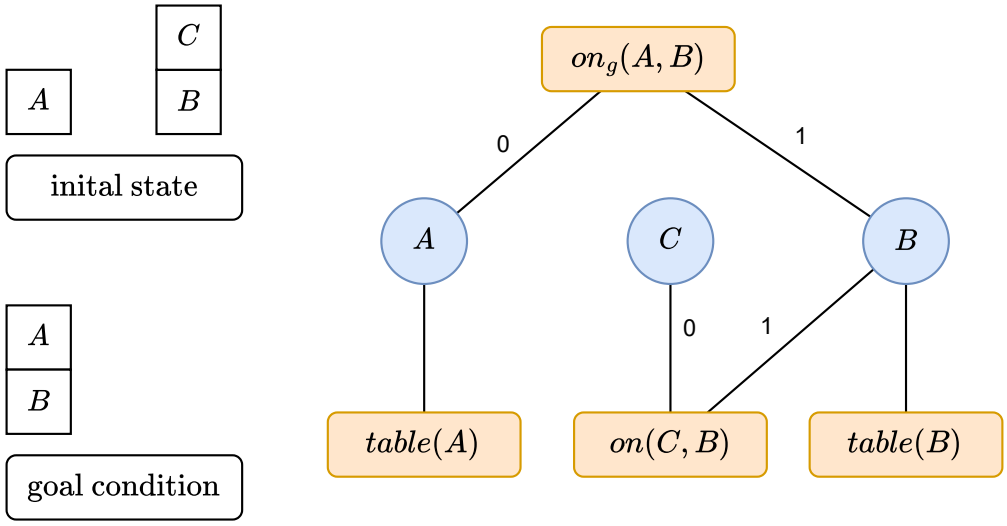


Figure 2.3: An example Blocksworld problem, given an initial state and goal condition, encoded as a relational graph. Circles represent objects, rectangles represent atoms, and the numbers on the edges indicate the position of the object within the respective atom. Example adapted from Chen and Thiébaux [8].

denote predicates, with $p_g \in Preds$ denoting goal-specific variants of predicates. Atoms that hold in a given state are written as $q = p(o_1, \dots, o_k)$, where k is the arity of the predicate p.

We define a hypergraph $G(Obj, E)$ with edge set

$$E = \{p^k \in Preds \cup Preds_g \cup Preds_{\neg g}\}.$$

Here, objects form the vertices, and the relational structure is induced by the instantiation of predicates over these objects. Unlike standard GNNs, which typically rely on binary relations, R-GNNs naturally support predicates of arbitrary arity: unary predicates correspond to arity 1, binary predicates to arity 2, and higher-arity predicates are also possible. Crucially, the ordering of objects within a predicate is preserved, ensuring that, for example, $on(A, B)$ and $on(B, A)$ are treated as distinct relations. This ordering is preserved explicitly. Goal and negative-goal predicates are modeled as separate relations.

We now turn to the message-passing procedure. As an initialization step, the embeddings of all object nodes are set to zero:

$$x_o^0 = 0^n, \quad \forall o \in Obj.$$

For each predicate $p \in Preds$ of arity k, we define three multi-layer perceptrons (MLPs) to handle message updates for the corresponding atoms:

$$\mathrm{MLP}_p, \quad \mathrm{MLP}_{p_g}, \quad \mathrm{MLP}_{\neg p_g}.$$

This setup allows a two-step message-passing algorithm: object nodes send embeddings to predicate-type nodes and then receive updated embeddings.

For an atom $q \coloneqq p(o_1, \dots, o_k)$ (ignoring goal encodings for simplicity), we first stack the embeddings of its objects:

$$h_q^i = \begin{bmatrix} x_{o_1}^i \\ \vdots \\ x_{o_k}^i \end{bmatrix}.$$

The predicate MLP is then applied to produce a message:

$$m_q^i = \mathrm{MLP}_p\big(h_q^i\big).$$

This message is unstacked into individual object messages:

$$m_q^i = \begin{bmatrix} u_{q,o_1}^i \\ \vdots \\ u_{q,o_k}^i \end{bmatrix}, \quad o_1, \dots, o_k \in q.$$

The messages u_{q,o_j}^i are sent back to the original object o_j and aggregated over all relations. They are then combined with the current embedding using an update MLP:

$$x_o^{i+1} = \mathrm{MLP}_{embedding}\big(x_o^i, \mathrm{LSE}\big(u_{q,o}^i \mid q \in \mathrm{Atoms}\big)\big) + x_o^i, \quad \forall o \in Obj.$$

After L rounds of message passing, a scalar output is obtained via global pooling followed by a readout function:

$$y = \mathrm{MLP}_{readout}\Big(\sum_{o \in Obj} x_o^L\Big).$$

This formulation corresponds to the R-GNN utilized by Ståhlberg et al.[38].

To illustrate, consider the classical planning problem “Blocksworld”. Objects correspond to blocks, the binary predicate on indicates stacking relations (e.g., block C is on block B), and the unary predicate table indicates if a block is on the table. A goal atom may specify that block A should be stacked on block B. This corresponds to the graph encoding illustrated in Figure 2.3.

2.5 Expressive power of GNNs

The expressive power of message-passing GNNs can be studied through their connection to fragments of first-order logic (FO), in particular $\mathcal{C}_2$ and $\mathcal{GC}_2$.

$\mathcal{C}_2$ is a fragment of first-order logic restricted to **two variables** and equipped with counting quantifiers. Counting quantifiers extend the existential quantifier by allowing constraints on the number of distinct variable assignments that satisfy a formula.

For example, the statement "there are at least three red blocks" can be written in $\mathcal{C}_2$ as:

$$\exists^{\geq 3} x \, \mathrm{red}(x).$$

This formula cannot be expressed in two-variable FO logic without counting quantifiers. In contrast, in standard FO we could express it as:

$$\exists x, y, z \, (x \neq y \wedge x \neq z \wedge y \neq z) \wedge \mathrm{red}(x) \wedge \mathrm{red}(y) \wedge \mathrm{red}(z),$$

which explicitly requires three variables. More generally, any counting quantifier can be unfolded into FO by introducing N distinct variables:

$$\exists^{\geq N} x \, \phi(x) \quad \equiv \quad \exists x_1, \ldots, x_N \bigwedge_{i \neq j} x_i \neq x_j \; \wedge \; \bigwedge_{i=1}^{N} \phi(x_i).$$

$\mathcal{C}_2$ is therefore strictly less expressive than full FO [4].

$\mathcal{GC}_2$ is an even more restrictive fragment of $\mathcal{C}_2$. Its formulas can only be built using:

$$\neg\varphi(x), \quad \varphi(x) \wedge \psi(x), \quad \exists^{\geq N} y \, (E(x, y) \wedge \varphi(y)),$$

where $\varphi, \psi \in \mathcal{GC}_2$. Intuitively, this means that:

- universal quantifiers are disallowed,
- quantification is limited to neighbors via the binary relation $E(x, y)$.

Since every $\mathcal{GC}_2$ formula is also in $\mathcal{C}_2$, but not vice versa, $\mathcal{GC}_2$ is strictly less expressive.

Barceló et al. [4] show (via a Weisfeiler-Lehman isomorphism argument) that the expressive power of message-passing GNNs lies between $\mathcal{GC}_2$ and $\mathcal{C}_2$. Concretely, any node classification problem expressible in $\mathcal{GC}_2$ can be captured by a GNN, given suitable parameters. If, in addition, each layer does global pooling, then this also holds for all formulas in $\mathcal{C}_2$.

For R-GNNs, the situation is subtler. Since R-GNNs perform graph classification (unlike [4], who only argued for node classification) with a pooling layer applied only at the final step, their expressive power is guaranteed to cover $\mathcal{GC}_2$ but may fall short of the full power of $\mathcal{C}_2$ [36]. That means, for every $\phi \in \mathcal{GC}_2$, an R-GNN can be constructed to satisfy ϕ, even in graph-level tasks.

Since we encode planning problems using R-GNNs, it is important to understand the consequences of their expressive power. Ståhlberg et al. [36] showed experimentally that R-GNNs fail

on some planning tasks whose optimal value function cannot be described in $\mathcal{C}_2$. Conversely, when they [36] found the value function to be expressible in $\mathcal{C}_2$, R-GNNs successfully learned it. This suggests that the effective expressive power of R-GNNs lies between $\mathcal{C}_2$ and $\mathcal{C}_3$.

To illustrate, consider a grid world where each cell is represented as an object, with unary predicates goal and player, and adjacency encoded by a binary relation $E(x, y)$. The optimal value function (the distance from the player to the goal) can be defined semi-logically as:

$$\begin{aligned} p_0(x) &= \text{goal}(x), \\ p_i(x) &= \exists y\ (E(x, y) \wedge p_{i-1}(y)), \\ sp_i(x) &= p_i(x) \wedge \neg p_{i-1}(x), \\ V^*(s) &= \sum_i i \cdot [\![\exists x\ (\text{player}(x) \wedge sp_i(x))]\!]. \end{aligned}$$

Here, $[\![\cdot]\!]$ is the Iverson bracket, which evaluates to 1 if the inner logical formula holds and 0 otherwise. Since these formulas are in $\mathcal{C}_2$, we expect an R-GNN to learn this value function, and indeed it can experimentally learn these graph distances.

However, the logical encoding of the planning domain is crucial. Ståhlberg et al. [39] considered a variant where cells are represented by their coordinates (x_i, y_i), with connectivity expressed by a quaternary relation $E(x_i, y_i, x_j, y_j)$. The shortest-path definition then requires at least four variables:

$$\begin{aligned} p_0(x, y) &= \text{goal}(x, y), \\ p_i(x, y) &= \exists x', y'\ (E(x, y, x', y') \wedge p_{i-1}(x', y')), \\ sp_i(x, y) &= p_i(x, y) \wedge \neg p_{i-1}(x, y), \\ V^*(s) &= \sum_i i \cdot [\![\exists x, y\ (\text{player}(x, y) \wedge sp_i(x, y))]\!]. \end{aligned}$$

This formulation belongs to $\mathcal{C}_4$, and indeed Ståhlberg et al. observed that R-GNNs fail when this encoding is used. Thus, the way we define the planning domain directly impacts whether it falls within the expressive scope of R-GNNs.

3 Related Work

Deep-learning–based methods have become increasingly prevalent in artificial intelligence research and have demonstrated strong performance compared to human-designed heuristics, as exemplified by AlphaGo in Go and AlphaZero in Chess [34]. Consequently, the field of classical planning has turned its attention toward integrating deep learning techniques into established planning algorithms. In this section, we review the progress of a promising approach that combines classical, logic-based, views of planning with reinforcement-learning–driven methods. We first discuss this method and its advances in applying deep learning to classical planning before focusing on Sokoban as a case study, since it represents a particularly challenging and for us relevant domain that has received attention even outside the context of classical planning.

3.1 Learning-Based Planning

A central challenge in applying reinforcement learning to planning is defining a suitable reward function. While in theory each state has a unique optimal cost-to-go value, this value is not readily available in practice. Simple reward functions, such as assigning -1 per action, result in sparse rewards that hinder learning. Gehring et al. [16] propose an alternative by employing planning heuristics, such as h^{FF} [23] or h^{add} [6], to provide denser and more informative reward signals.

3.1.1 Supervised Value Learning

A promising attempt relevant to our work is the application of graph neural networks (GNNs) in planning introduced by Ståhlberg et al. [36], who proposed the R-GNN model. This architecture encodes planning states and goals as graph representations. For training they are using a value-based loss on the optimal value function, $L_2(v^{gnn}, v^*)$. To generate training data, they executed random actions from the initial state to obtain novel instances, which were then solved using admissible search. Optimal solution paths $(s_0, \dots, s_n)$ are added to the dataset as state-value pairs (input $= s_i$, target $= n - i$). Their results demonstrated high success rates with near-optimal plan quality and strong generalization results, albeit on relatively simple domains. Expecially the reliance on external optimal solvers to generate a fixed-size trainings dataset (40,000 states per domain) limited scalability.

3.1.2 Unsupervised Value Learning

Building on this foundation, Ståhlberg et al. [37] later refined their approach by introducing a ranking-based loss function. They argued that R-GNN training may fail to converge, particularly when optimality is NP-hard, as in domains such as Blocks or Reward (a grid-based domain where collecting rewards optimally reduces to solving a traveling salesman problem). They introduce a ranking-based loss called L_1 which enforces the inequality

$$V(s) \geq 1 + \min_{s' \in N(s)} V(s'),$$

ensuring that executing a greedy policy is sufficient to solve the domain. While this method showed promising results on selected tasks, it still required access to the optimal value function, thereby restricting its application to relatively small, solvable problems. A related perspective was later provided by Chen et al. [8], who applied a similar ranking-based objective to numerical planning. In contrast to Ståhlberg's agent-centric use of the value function, Chen et al. employed the learned function as a heuristic for greedy best-first search, highlighting different ways of leveraging learned knowledge.

3.1.3 Actor–Critic Approach

Recognizing the limitations of relying on global value functions, Ståhlberg et al. [38] extended their research trajectory with a policy-gradient method based on an actor–critic architecture. This formulation enables learning without depending on value-based or ranking-based methods at inference time, instead directly learning policies over transitions $\pi(s \mid s')$. Similar to the ranking-based approach, it requires only local properties rather than global consistency across the entire domain. Training was conducted in a one-step reinforcement learning setting, but it required sampling from near-goal states, which restricts applicability to state spaces that can be reasonably expanded.

Stante et al. [40] adapted this method to a more general model-free reinforcement learning setting using a replay buffer and full rollouts. However, they still faced the persistent challenges of sparse rewards in large state spaces and the inherent difficulties of stabilizing and training actor–critic approaches.

3.1.4 Expressivity

Throughout these iterations, Ståhlberg and colleagues also emphasized the expressivity limits of their R-GNN models. The original architecture is experimentally bounded by $\mathcal{C}_2$ expressivity, constraining its applicability across planning domains. To address this, the authors introduced R-GNN[t] [39], which enhances expressivity by preprocessing domains to introduce new objects and predicates. For instance, pairs of objects (o_1, o_2) are combined into new composite objects $(o_1 o_2)$, allowing the model to capture relations that were previously inaccessible in $\mathcal{C}_2$. While

this extension broadened the applicability of R-GNNs, it did so at significant computational and memory costs, as its complexity grows quadratically in the number of objects. This highlights a broader research tension: while architectural extensions can enhance expressivity, more efficient alternatives, such as enriching the planning domain itself with derived predicates, may achieve similar benefits with lower overhead.

Taken together, these works trace a clear trajectory: beginning with supervised learning on solver-generated data [36], moving toward ranking-based relaxations to address NP-hardness limits [37], advancing to reinforcement-learning formulations with actor–critic methods [38], and finally exploring architectural expressivity through domain preprocessing [39].

Despite these advances, scalability and sparse rewards remain central challenges in applying deep learning techniques to more complex classical planning domains such as *PushWorld*.

3.2 Deep Learning in Sokoban

Since *PushWorld* [25] has not yet been widely studied, we turn to *Sokoban* as an important precursor and source of inspiration. *Sokoban* ("warehouse keeper" in Japanese) is a grid-based combinatorial puzzle game originally released in 1982 by Hiroyuki Imabayashi. It exhibits many of the same challenges as *PushWorld*, most notably sparse rewards and combinatorial complexity. The decision problem for *Sokoban* is PSPACE-complete [9, 10, 21], even without considering optimality, which already renders the approximation of a generally correct value function intractable. Sparse rewards further exacerbate the difficulty. Nevertheless, researchers have risen to the challenge and attempted to address this problem.

3.2.1 Curriculum Learning

Feng et al. [12, 13] mitigate the sparse-reward issue using curriculum learning. Their method generates simplified Sokoban instances by randomly removing boxes and goals, allowing agents to progress from easily solvable, small-scale problems to increasingly difficult ones. A bandit algorithm adaptively selects training problems suited to the agent's current skill level. This strategy effectively enables learning on instances with up to 15 goal boxes. However, the approach requires a meaningful notion of relaxing difficulty, which is less straightforward in domains like *PushWorld*, where the presence or absence of a single object can drastically alter complexity. Additionally, they made use of search-based training and execution methods akin to AlphaZero [34].

3.2.2 Model-Free Methods

Some works deliberately avoid domain knowledge, focusing on purely model-free methods. Guez et al. [18] introduce the Deep Recurrent Convolutional (DRC) model, which combines convolutional inputs with an internal LSTM [22] and separate policy and value heads. Using an

actor–critic REINFORCE framework [45], they achieve a coverage of 99% on 10×10 Sokoban instances with four goals. The original approach to solving Sokoban, however, employed a hand-crafted reward function (e.g., penalties for steps, rewards for pushing boxes onto goals), which helps with the sparse-reward setting but still required manual domain effort.

Internal Planning

In their approach, Guez et al. [18] highlight that additional recurrent iterations at test time improve solution quality, suggesting that the model performs implicit lookahead akin to planning. Follow-up studies [7, 42] further investigated this phenomenon and reached the same conclusion: the model can greatly benefit from additional computation at test time, and there is evidence that it constructs internal plans mimicking a search process.

Bansal et al. [3] explore a related idea through "deep thinking" architectures, which investigate whether neural agents can exploit additional test-time computation. While primarily applied to maze-solving, this line of work has implications for recurrent architectures such as DRC [18] or R-GNN [36]. A limitation is that their evaluation focuses on zero-shot generalization rather than reinforcement learning.

3.2.3 Search-Based Methods

An interesting contribution from Orseau et al. [31] investigates the combination of a pre-trained actor–critic agent with tree-based search. They present both theoretical and experimental results on the number of nodes expanded during execution. Notably, their comparison to the LAMA solver shows significantly shorter solution paths while expanding only about twice as many states. They successfully solved all of the test problems they defined.

Additional relevant works include those of Agostinelli et al. [2, 29], who introduced DeepCube and DeepCubeA for solving the Rubik's Cube. These methods address sparse rewards by generating training data through reverse random walks: starting from the goal state, they apply $t \sim U(1, K)$ scrambling steps, yielding states sampled at controlled distances from the goal. As training progresses, K increases, exposing the model to harder problems. Distances are learned by one-step rollouts updating both policy and value functions. During execution, DeepCube uses MCTS guided by the learned policy [29], while DeepCubeA employs A* with the learned value function [2]. Applied to Sokoban, these approaches achieve strong results, outperforming classical planners such as LAMA in both solution quality and efficiency. Nonetheless, they rely heavily on access to the goal state and on the assumption that reverse random walks produces a meaningful training distribution. Assumptions that do not hold universally.

3.3 Summary

Sokoban remains a challenging reinforcement learning benchmark, with the most successful methods combining search and learning, as seen in approaches inspired by AlphaZero and informed A*. Many existing solutions rely on convolutional inputs, which limit generalization. By contrast, relational graph neural networks (R-GNNs) offer relational inductive biases and invariances that promise better generalization. Motivated by these advances and inspired by search-augmented learning frameworks, our work explores R-GNN-based approaches to *PushWorld*, aiming to bridge this gap.

4 Methods

In this section, we present our approach in a structured manner. We examine the current limitations of R-GNNs and learning-based planning methods. First, we discuss a scalable, planner-free training strategy that allows us to train in domains much larger than those that can fit in memory. Next, we address the constraints imposed by layer size in R-GNNs, which hinder the ability to capture long-range dependencies. We also review our empirical findings on generalization and propose two methods to mitigate the identified challenges, aiming to improve overall generalization performance.

We then outline the domains of interest, describe how they are logically encoded in PDDL, and propose an adapted PDDL encoding for *PushWorld* that is more aligned with our learning-based planning approaches. Finally, we focus on expressivity and the challenges of $\mathcal{C}_2$ in R-GNNs, and explore how derived predicates can be leveraged in ways that may transfer effectively to other domains.

4.1 AV* Training Method

Scaling learning based planning algorithms to handle large and combinatorially complex domains such as *PushWorld* presents unique challenges. Learning based planning techniques typically rely on the complete state-space or a perfect planner. While these methods perform well on small instances, their scalability is limited. The exponential growth of the state space due to increasing grid size and number of objects makes exhaustive search infeasible, even for moderately sized problems.

In some planning domains, it is possible to learn the core logical concepts from small instances and use that to generalize to larger ones, as showcased by Ståhlberg et al. [36]. However, this is not the case for *PushWorld*. The complexity of this domain is not merely due to the number of states but stems from deep structural dependencies that only emerge at scale. For instance, box-pushing interactions and deadlock configurations in small 4×4 grids differ qualitatively from those in larger environments with multiple boxes, goals, and obstacles. Solving large *PushWorld* instances requires reasoning about long-term dependencies and interaction effects that are absent in small examples.

Reinforcement learning (RL) presents an avenue for scaling, but it faces significant drawbacks in this setting. Reward signals are typically only available upon reaching a goal, resulting in weak signals, especially in long-horizon / needle in the haystack tasks like Sokoban. That is

why many RL-based planning approaches rely on access to a dense reward function or require extensive precomputation of value targets.

To overcome these limitations, we propose a scalable hybrid method that integrates model-based planning with deep approximate value iteration [2]. Our approach leverages the deterministic and fully observable nature of classical planning problems. In such settings, each state has a well-defined minimal cost-to-go: the minimum number of steps required to reach the nearest goal state. Rather than attempting to learn a policy directly or rely on terminal rewards, we focus on learning this cost-to-go function, which serves as a powerful guiding signal for planning.

4.1.1 Learning through Value Estimation

In deterministic and fully observable environments like *PushWorld*, the optimal planning problem is equivalent to finding the shortest sequence of actions from an initial state to a goal. This allows us to frame learning as value estimation: for each state, we seek to learn its true cost-to-go. If we had access to the optimal cost-to-go function $v^*(s)$, planning would then reduce to greedy minimization over successor values.

We introduce a planning-based learning algorithm inspired by A* search and value iteration. The key idea is to use a learned value function $V(s)$ as a heuristic to guide search, and then use the outcome of this search to further refine $V(s)$. Specifically, we employ a weighted A*-like expansion procedure, where node priorities are determined using the sum of current search depth and predicted value. This search yields a tree of expanded states, which we convert into training data for the value function in an offline RL-like fashion.

To approximate the value function, we use a Relational Graph Neural Network (R-GNN) with a value based readout head, which naturally handles the relational structure of the domain. After each planning rollout, the search tree is translated into supervision targets for the R-GNN, enabling continual refinement of the heuristic. By alternating between planning and learning, our method bootstraps from initially poor approximations toward increasingly accurate value estimates, without requiring ground-truth cost-to-go labels, handcrafted heuristics, state-of-the-art classical planners, or dense reward signals.

4.1.2 Search Procedure and Value Updates

Our learning method alternates between searching and value function updates. The search component is using A*, where nodes (states) are prioritized based on a combination of their current search depth and their estimated cost-to-go, as predicted by the learned value function $V(s)$. This guides the search toward promising regions of the state space without requiring explicit goal distance pre-computations.

If a goal state is encountered during search, we extract the solution path from the initial state and use it to assign target values to visited states. If no goal is found within the search budget,

we still use the explored search tree to propagate value estimates through local bootstrapping using full bellman updates.

The value function is updated based on the following two principles:

- **Full Bellman updates for expanded nodes:** For every expanded node s, we update its value using the bellman update:

$$V^{n+1}(s) = \max_{s' \in N(s)} (V^n(s')) - 1$$

  where $N(s)$ denotes the set of successor states reachable from s. This update reflects the property that any state must be at least one step farther from the goal than its most promising successor. It is consistent with the Bellman update equation used in value iteration, and it also holds for the true cost-to-go function $v^*(s)$.
- **Successful path bounding:** When a solution path is found, we can directly bound the value function along that path. Let the path be $(s_m, s_{m-1}, \dots, s_0)$, where s_0 is the goal state and s_m is the root of the search tree (the initial state). We assign:

$$V^{n+1}(s_i) = \max\big(V^{n+1}(s_i), -i\big)$$

  for each $i \in \{0, \dots, m\}$. This directly bounds the approximated value function along the discovered path. While it would theoretically suffice to label only the goal state $V(s_0) = 0$, propagating these exact values helps accelerate convergence during learning, especially early in training. Notice that we cannot say $V^{n+1}(s_i) = -i$ as the found goal path is not guaranteed optimal and this could potentially lead to a worse solution.

In cases where no solution is found within the search budget, the Bellman updates still enable value propagation through the explored subtree. This allows the model to gradually refine its estimates, lowering the value of unpromising branches and thus implicitly encouraging the search to shift toward more promising regions in future episodes.

4.1.3 Algorithm

We summarize our method in Algorithm 2, referred to as **AV***. The algorithm performs a value-guided search, extracts training data, and uses it to iteratively improve the value function. Once the R-GNN has been trained on the generated data, running this data acquisition algorithm produces different search results until convergence.

AV* has several key hyperparameters: the search budget, the ϵ-greedy exploration factor, the step size, and the dead-end cost. The search budget limits the number of states visited during the search. A higher search budget yields more training data but also increases computational cost. The default search budget is set to 2048. While a larger budget can be beneficial, early in training many problems are extremely difficult and would require an impractically large

budget to solve. After sufficient training, the policy becomes accurate enough to prune many states, allowing smaller search budgets to suffice even for difficult problems.

The ϵ-greedy exploration factor controls the fraction of newly discovered states that are added to the beginning of the priority queue, encouraging exploration. This mechanism prevents the value function from overestimating certain states and forces the policy to reconsider them. In our experiments, ϵ is set to 0.05, which is relatively low. The step size determines the weight assigned to the distance from the initial position. A step size of 0 reduces AV* to a best-first search, while a step size of 1 corresponds to a classical A* search. Here, the step size is chosen as 0.5, so both the value estimation and the steps taken so far influence the search.

Finally, the dead-end cost defines the lower bound for value function predictions, i.e., $[v_{\text{dead-end}}, 0]$. We set $v_{\text{dead-end}} = -200$, ensuring that no prediction falls below this threshold. Assigning a dead-end value provides a stable numeric target for states from which no further action is possible. In standard reinforcement learning, this is implicitly handled via the discount factor γ, but since we use $\gamma = 1$ for interpretability, we explicitly define a dead-end value instead.

4.1.4 Theoretical Properties

We provide a theoretical guarantee that the value function converges to the optimal cost-to-go function in the tabular case. The proof is similar in spirit to the one for value iteration.

Theorem 1. *If the algorithm is run indefinitely ($\lim_{t\to\infty}$) with a tabular value function $V^t(s)$, and assuming initial values $V^0(s) = -\infty$, it will eventually converge to the optimal value function $v^*(s)$.*

Proof (by induction). Initially, if no goal path is found, we have

$$V^1(s) = V^0(s) = -\infty.$$

Assume $t = 0$ is the first time step when a goal path is found. For all non-goal-path states, $V^1(s) = V^0(s) = -\infty$. For states on the goal path $(s_n, \dots, s_0)$, where s_n is the initial state and s_0 the goal, the path-bounding update gives

$$V^1(s_i) = \max\big(V^0(s_i), i\big) \geq V^0(s_i), \quad \forall s_i \text{ on the path.}$$

Now assume as the induction hypothesis that

$$V^t(s) \geq V^{t-1}(s), \quad \forall s.$$

Algorithm 2 AV* (Search with Learned Value Function)

```
1  procedure AV*(problems, state_embedder)
2      Select a random problem from problems
3      init ← Initial state of problem
4      visited ← {init}
5      predecessors ← {init : None}
6      value, distance ← ∅, {init : 0}
7      Initialize priority queue queue with init
8      goal ← None
9      while queue not empty and |visited| < 2048 do
10         cur ← pop highest priority from queue
11         if cur satisfies goal condition then
12             goal ← cur; break
13         actions ← applicable actions at cur
14         if actions = ∅ then
15             value[cur] ← −200.0                          Dead end penalty
16             continue
17         next_states ← successors of cur
18         values ← R-GNN estimate of V for next_states
19         value[cur] ← clip([−200, 0], max(values) − 1)    Clipped bellman updates
20         for n in next_states do
21             if n ∉ visited then
22                 Add n to visited, set predecessors[n] = cur,
                   distance[n] = distance[cur] + 0.5
23                 priority ← distance[n] − value[n] A* heuristic, negative values as cost is neg-
                   ative
24                 if rand() < 0.05 then
25                     priority ← 0                         ?Exploration
26                 Insert n into queue with priority
27     if goal ≠ None then
28         tmp ← goal; v ← 0
29         while tmp ≠ None do          Bounding of successful paths to bootstrap learning
30             value[tmp] ← max(value[tmp], −v)
31             v ← v + 1; tmp ← predecessors[tmp]
32     return value.keys(), value.values()
```

Then for the next iteration:

$$V^{t+1}(s) = \max_{s' \in N(s)} V^t(s') - 1 \geq \max_{s' \in N(s)} V^{t-1}(s') - 1 = V^t(s),$$

so the sequence $V^0(s) \leq V^1(s) \leq V^2(s) \leq \ldots$ is monotonically non-decreasing.

Upper bound. Next, we show that $V^t(s)$ is upper bounded by $v^*(s)$. Suppose there exists a minimal t and a state s such that

$$V^t(s) > v^*(s).$$

Then either

$$V^t(s) = \max_{s' \in N(s)} V^{t-1}(s') - 1,$$

so there exists some s' with

$$V^{t-1}(s') - 1 > v^*(s).$$

But since v^* satisfies the Bellman equation, we must also have

$$V^{t-1}(s') - 1 > \max_{s' \in N(s)} v^*(s') - 1 \quad \Rightarrow \quad V^{t-1}(s') > v^*(s').$$

This contradicts the minimality of t.

Alternatively, s lies on a goal path, but in that case the path-bounding update cannot exceed the optimal path length. Both possibilities lead to a contradiction.

Strict improvement. Finally, consider a state s with $V^t(s) \neq v^*(s)$, whose optimal next state is s' such that $V^t(s') = v^*(s')$. Due to the non-zero exploration probability ϵ, the algorithm will eventually expand a path that reaches s, giving

$$V^{t+1}(s) = \max_{s' \in N(s)} V^t(s') - 1 = v^*(s') - 1 = v^*(s),$$

so that $V^{t+1}(s) > V^t(s)$.

Such a state s must exist because we assume $V^t(s_g) = 0$ for all goal states s_g. If $V^t(s_g) \neq 0$, then, since a path exists to s_g, the algorithm will eventually expand s_g due to the non-zero exploration probability ϵ, after which $V^t(s_g) = 0$ holds. This guarantees that there is at least one state s with $V^t(s) \neq v^*(s)$ whose value will strictly improve.

Combining monotonicity, upper bound, and occasional strict improvement, the theorem is proved.

4.1.5 Greedy Property

While Theorem 1 guarantees convergence in tabular settings, in practice we approximate $V(s)$ using a neural network (R-GNN). Representation limits may prevent exact convergence, especially in PSPACE-complete domains like Sokoban. Nevertheless, the primary goal is not a perfect cost-to-go function, but reliable greedy goal-reaching behavior.

We desire the following property: for a successful trajectory from initial state to goal $(s_m, \dots, s_0)$,

$$\forall s_i \text{ on the path}, \quad s_{i+1} = \arg\min_{s' \in N(s_i)} V(s').$$

That is, greedily following the learned value function should reach the goal. Empirically, this holds on training instances after sufficient learning. It naturally follows if $V(s) = v^*(s)$.

4.1.6 Blocksworld Example

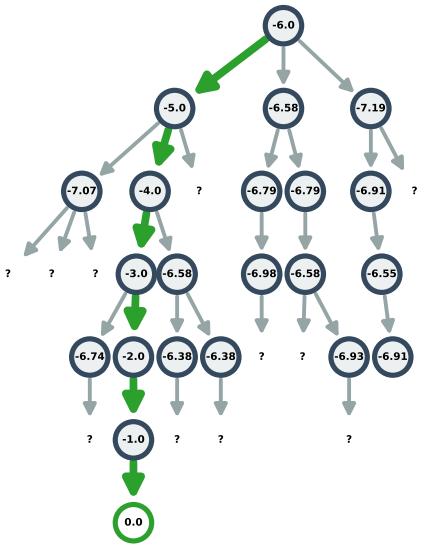
To illustrate how the algorithm functions, we train a R-GNN on a simple Blocksworld instance consisting of four blocks using AV*. Figure 4.1 shows the AV* search tree and corresponding value targets at the beginning, middle, and end of training. States that were not expanded but whose values were computed during search are marked with "?" since no new targets were assigned to them. At the start of training, the R-GNN is untrained and has just been initialized. As a result, the search tree is fully expanded and all target values are nearly identical. This illustrates the limitation of pure Bellman updates: only the states along the found path have meaningful updated targets, while all other values remain inaccurate. By the midpoint of training, the model has begun to identify promising paths with reasonable confidence. Many branches that do not lead to the goal are pruned, though several leaf nodes remain partially expanded. Value estimates are improving but are not yet fully accurate. At the end of training, the model demonstrates the greedy property: it expands only the nodes along the shortest path to the goal. The search tree becomes highly efficient, with no unnecessary expansions, showing that the model has successfully learned the value function.

4.1.7 Training Architecture

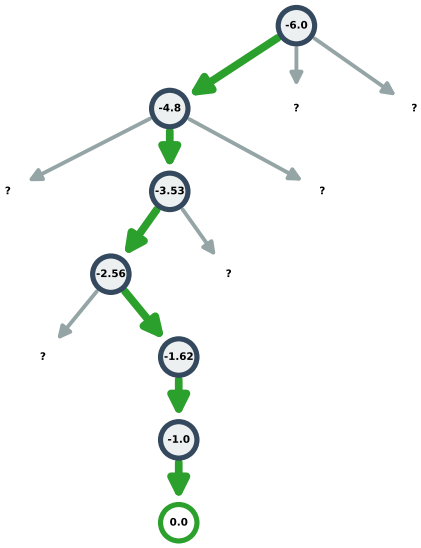
Working with graphs, as logical representations of a state in the state space, can be challenging. A significant portion of CPU compute is spent converting from the logical state to its corresponding relational graph representation. This motivated the adoption of a distributed training approach, as illustrated in Figure 4.2.

Our system consists of two primary types of processes: a trainer and multiple workers. The trainer is responsible for optimizing the model using the data provided by the workers. This is an iterative process, as workers also require the model weights to generate improved data. The loop continues until the model converges.

(a) Start of training



(b) Midpoint of training



(c) End of training

Figure 4.1: AV* search trees for a single Blocksworld problem at different stages of training. The initial state is the tree root. States that were not expanded but whose values were computed are marked with a ? since no new targets were assigned to them. The goal path is highlighted with a thicker green line, and the goal node is also outlined in green. Node values represent the value targets for that state.

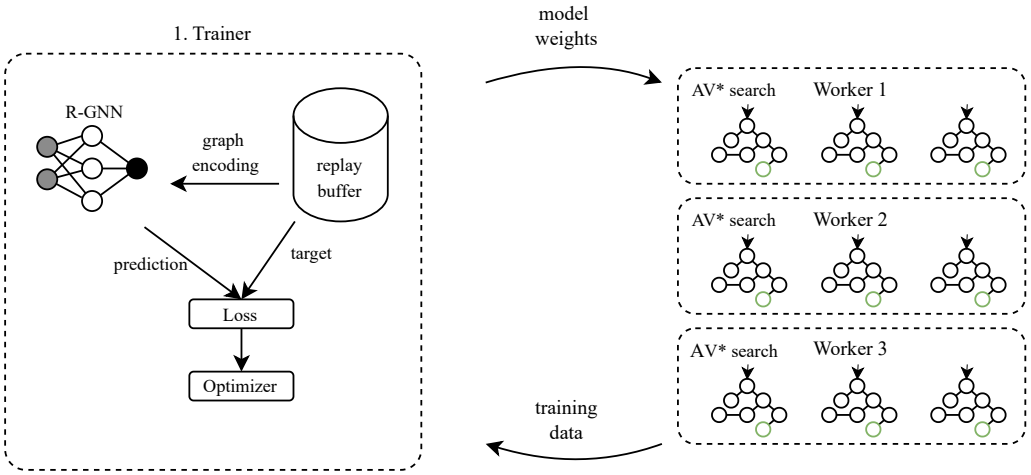


Figure 4.2: Training architecture illustrating the distributed training loop for generating data and optimizing the model.

By delegating the computationally intensive tasks of state generation, graph construction, and data batching to the worker processes, we ensure that these CPU-bound tasks are efficiently parallelized, maintaining high GPU utilization. Additionally, all data generated by the workers is stored in a FIFO-style replay buffer, a common approach in reinforcement learning algorithms such as deep Q-Learning (DQN). This allows training to continue even when no new data is available.

To prevent overfitting, the main training loop is executed only when new data is received, though previously collected data is reused. Furthermore, each worker performs multiple runs of the AV* algorithm concurrently. This batching strategy improves GPU utilization: with only a single AV* search tree, the GPU would be underutilized, as it would only process a few next states at a time. Running multiple AV* searches simultaneously allows the pooling of value estimation requests, thereby fully leveraging the GPU.

Weights are shared between the trainer and the workers periodically. This has a two fold effect, on the one hand the data generation process is generating data and targets using delayed weights, similar to how its done in DQN, ensuring stability. Every worker produces 20 batches of 128 states, after which they fetch the newest version of the model parameters and continue producing training batches.

The general training loop on the main process is sketched in Algorithm 3.

4.1.8 Instance Difficulty and Training Sampling

Since our approach is more closely aligned with reinforcement learning (RL) than previous learning-based methods, we require a significantly more diverse set of problem instances.

Algorithm 3 Training Loop

```
1 while training do
2     Let (x, y) be a training batch generated by AV* (Algorithm 2), where x represents states
      and y the corresponding target values.
3     Wait for a new training batch from one of the workers
4     Add (x, y) to the buffer
5     for each batch (x_b, y_b) in the buffer do
6         out ← R-GNN(x_b)
7         Compute L = loss(out, y_b)
8         Backpropagate ∇L
```

Unlike prior work [36], we cannot simply enumerate the state space and randomly sample a state to use as a new starting point. In particular, random walks are not effective: a single incorrect action can render a problem unsolvable, even though it may initially appear solvable. For example, pushing a box against a wall may make the task unsolvable while still generating many new states to explore.

To address this, we train on a much larger set of problem instances, often consisting of thousands of unique tasks. Consequently, the sampling method of problems during training becomes important. Assuming no prior knowledge of each problem's difficulty (a reasonable assumption when scaling to problem sizes that classical planners can no longer solve), we must carefully design a sampling strategy. If the AV* algorithm is applied to problems that are already easily solvable, it does not produce a meaningful learning signal. However, occasionally sampling these easier problems is still beneficial, as it prevents the model from overfitting to harder tasks, and helps retain performance on simpler ones. Random sampling initially treats easy and hard problems equally, which is acceptable at the start, but once only a few hard problems remain, selecting these harder problems becomes unlikely. Therefore, a more sophisticated sampling strategy is advantageous.

We observe that the difficulty of a problem can be estimated using results from the AV* algorithm via the following metric:

$$\text{diff}(p) = 1 - \frac{|\text{solution\_length}(p)|}{|\text{visited}(p)|} + \epsilon$$

Intuitively, if many branches contain many nodes, the heuristic is poor and the problem is hard; conversely, a better heuristic reduces the branching factor, making the problem easier. Sampling problems in proportion to this difficulty score biases the training process toward harder problems while reducing samples of easier ones. This metric also has a desirable scaling property: if a problem is not solved, we can assign it a fixed value of ϵ, ensuring that solvable problems are prioritized. Over time, as training progresses, the weights of problems dynamically adjust. If the total added weight across all problems converges to $\epsilon \cdot \text{len}(p)$, it

indicates that we can stop the training. This approach provides a dynamic measure of both problem difficulty and training progress.

4.2 Layer Size

One of the key limitations in what can be logically expressed, even within the expressivity of $\mathcal{C}_2$, is imposed by the layer size L. This parameter determines the number of message-passing iterations performed by the model. It is evident that with $L = 1$, the model lacks the capacity to learn meaningful structural relationships, such as grid distances between objects, as such information cannot be effectively propagated. Consequently, this limitation significantly affects the model's ability to generalize. Even if the model is capable of generalization, by having learned the correct logical value function, it remains constrained by the chosen value of L. In practice usually $L = 30$ is choosen empirically. This is because for our small problems this value is sufficient without introducing many of the problems that come with bigger L.

A seemingly straightforward solution would be to increase L to a very large value. However, this approach is impractical due to challenges such as numerical instability, exploding gradient norms (which impede learning), and the well-known issue of oversmoothing [28], particularly prevalent and well researched in GNNs. To address this challenge, and to explore whether we can mitigate, extend, or potentially eliminate the dependence on this hyperparameter L, we must first quantify the severity of the issue. To this end, we present selected results from Section 5.3.1 ahead of the main results. We examine the behavior of the gradient norm and variance and the effects of oversmoothing on our model trained with AV*. These preliminary findings are highlighted early to motivate the discussion of subsequent ideas and insights.

4.2.1 Embedding Variance and Norm Analysis

As a preliminary step, we introduce a simple metric to estimate the variance of the node embeddings across layers of message passing. If at some point the variance flattens out or gets close to zero this would mean that we are in fact having issues with oversmoothing. If the norm increases exponentially we might be dealing with numerical instability (as values grow larger so do the gradients leading to a worse performance or running into nan's if we increase the layer size).

Let x_o^l, with $o \in \text{Obj}$, denote the embedding of object o after l layers of message passing. We define the mean embedding at layer l as:

$$\text{avg}^l = \frac{1}{|\text{Obj}|} \sum_{o \in \text{Obj}} x_o^l.$$

Using the avg^l, we compute the variance of the embeddings at layer l as:

$$\text{variance}_o^l = \left\| x_o^l - \text{avg}^l \right\|^2$$

$$\text{variance}^l = \frac{1}{|\text{Obj}|} \sum_{o \in \text{Obj}} \text{variance}_o^l.$$

In addition to the variance, we also investigate the norm of the node embeddings, which we define as the Euclidean norm of E_i^l:

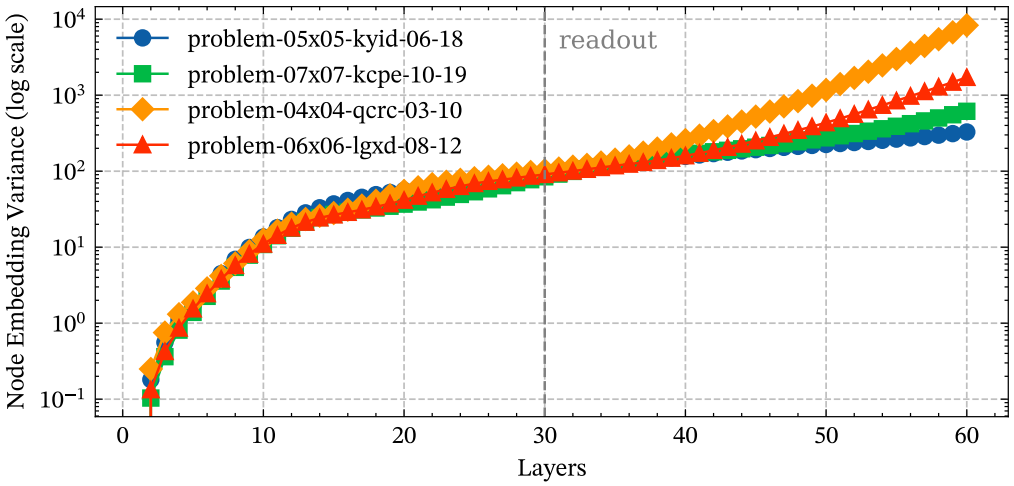
$$\text{norm}_o^l = \left\| x_o^l \right\|_2$$

$$\text{norm}^l = \frac{1}{|\text{Obj}|} \sum_{o \in \text{Obj}} \text{norm}_o^l.$$

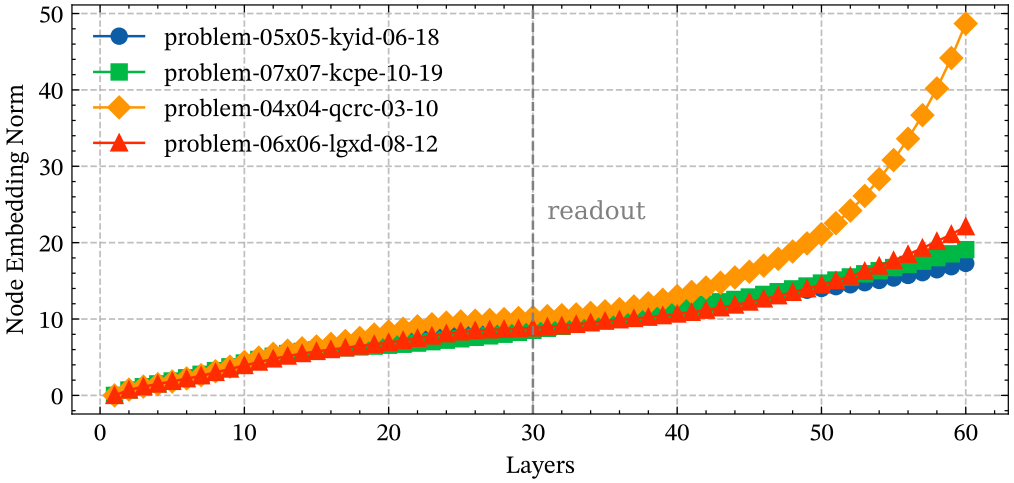
To gain insights into the behavior of these quantities throughout the layers, we plot both the variance and the mean norm across layers for a selection of problem instances from the Pushkoban (problem set introduced in the following sections) training set.

The results are shown in Figure 4.3. We can clearly see that we do not encounter issues with either oversmoothing or norm explosion. Even at the 30-layer readout, the values remain bounded and within a healthy range. It is particularly interesting to observe how the model's variance changes across layers. In many cases (though not all), it follows a roughly monotonic trend that introduces more variance. Examining the norms provides an even clearer picture: with each layer, the values generally increase, which in turn contributes to the variance growth. Additionally, it is noteworthy that the model initially attempts to constrain the embeddings, keeping them relatively similar until the readout. Afterward, the embeddings are left unbounded and continue to grow.

This is already a strong indication that, in theory, nothing fundamentally limits us from increasing the layer size. Many of the common issues and their usual remedies do not apply to our use case with GNNs. One possible explanation is that we are working with highly structured, small graphs and performing graph-level tasks (graph regression) rather than node classification. Additionally, our graph design starts with zero variance and incrementally adds information through the relational connections and predicate MLPs. This approach differs from others that begin by encoding node features, which are then mixed and can possibly converge to the same embedding. In our method, the problem input, which is its structure, is continuously reinforced with each round of message passing.



(a) Layer-wise variance.



(b) Layer-wise embedding norm.

Figure 4.3: Layer-wise variance and embedding norm of the R-GNN on four Pushkoban problems from the training set. The model corresponds to the model presented in the first row of Table 5.2.

4.2.2 Progressive Training

Inspired by Bansal et al. [3], we adapt a progressive training algorithm. The core idea in their work was to train a recurrent neural network (RNN) to solve maze-like problems. Their approach involved early stopping of RNN iterations at random time steps during training. This enforced that the agent maintained a valid intermediate solution at every iteration. At test time, they could then scale up the number of RNN iterations, effectively giving the model more ”compute,” which led to improved performance.

Additionally, they incorporated skip connections by feeding the original input into each RNN block. This prevented the model from losing critical information about the original problem during the iterative updates.

However, an important distinction must be made when adapting their approach to our setting. Their model operates in a one-step prediction regime, predicting the entire solution to a maze in one forward pass (e.g., an image of the full path). In contrast, our model is situated within the reinforcement learning paradigm, where the task is to predict the value function (cost-to-go) from a given state, not a complete trajectory or plan.

Nevertheless, inspired by their method, we adopt a similar progressive training approach, which allows us to vary the parameter L at test time. This variation could potentially lead to improvements in predictive accuracy. Unlike their setting, we do not need to reintroduce the original input at each layer, since our graph structure inherently encodes the input information. Furthermore, due to the nature of our model, such reintroduction is not structurally feasible.

The training strategy is as follows: for each batch, we randomly sample an integer layer depth $L \sim \mathcal{N}(30, 10)$, and select a secondary value $L_{\min} \in [0, L)$. We split the GNN message passing into two segments: a non-gradient (frozen) pass from layer 0 to $L_{\min}$, and a gradient-enabled pass from $L_{\min}$ to L.

The pseudocode is given in Algorithm 4.

Algorithm 4 Training with progressive loss

Let (x, y) be a training batch generated by AV^* (Algorithm 2), where x represents states and y the corresponding target values.
Sample $L \sim \mathcal{N}(30, 10)$
Sample $L_{\min} \sim \text{Uniform}(0, L)$
`no_grad()`:
    `embeddings` $\leftarrow$ R-GNN$_0^{L_{\min}}(x)$
`out` $\leftarrow$ Readout$\Big($R-GNN$_{L_{\min}}^{L-L_{\min}}($`embeddings`$)\Big)$
Compute $\mathcal{L} = \text{loss}(\texttt{out}, y)$
Backpropagate $\nabla\mathcal{L}$

This training procedure encourages the model to produce useful intermediate representations, while also allowing test-time scaling via increased depth L. By not propagating from the start we also bound the gradients to prevent numerical instability with to large gradients.

4.3 Generalization

In many grid-based domains such as *Reward*, *Rovers*, *Grid*, and *Visitall*, R-GNN models appear to struggle. In the work on expressivity by Staahlberg et al. [39], it was shown that the standard R-GNN achieved only 45% accuracy on *Rovers* and *Grid*, and 82% on *Visitall*. These results are considerably worse than those reported in an earlier publication [36], where most other domains achieved nearly a 100% solve rate on the test set.

A similar pattern emerges in our Pushkoban (introduced in the next section) experiments. This raises the question: why does the model generalize poorly in grid-based domains, while generalization in other domains appears much easier?

Of course, our notions of "easy" and "hard" here are not rigorous and are primarily based on empirical observations. The goal of this section is to investigate why these models fail to generalize and to explore potential solutions. One factor that could play a role is that the number of objects (cells) grows quadratically with grid size; for example, $19 \times 19 = 361$ cells in our largest grid. This could be a differentiating factor compared to comparatively scaling in other domains.

Since the *PushWorld* problem is quite complex, we first focus on a simplified version: a single-agent navigation problem on a grid, where the agent must reach a goal position. We train the agent on randomly generated goals and initial positions on grids of size 4×4 to 9×9 and test on a generalization set of 10×10 to 19×19. This simplification removes potential confounding factors such as arguments about expressivity limits or the PSPACE-hard nature of *PushWorld*.

For this simple problem (referred to as *Mouse*), an optimal $O(1)$ algorithm exists: compute the Manhattan distance between the agent's cell and the goal cell. While this is less trivial to express in logical form, it can be done with the following formula in $O(n)$:

$$\begin{aligned} p_0(x) &= goal(x), \\ p_i(x) &= \exists y \, (adjacent(x, y) \wedge p_{i-1}(y)), \\ V^* &= \sum_{i=1}^{L} i \cdot [\![\exists x \, (has\_goal\_box(x) \wedge p_i(x) \wedge \neg p_{i-1}(x))]\!] \end{aligned} \tag{4.1}$$

Here we use the Iverson bracket $[\![\phi]\!]$, which equals 1 if ϕ holds and 0 otherwise. This computes the shortest path in $\mathcal{C}_2$ for any path up to a length of L.

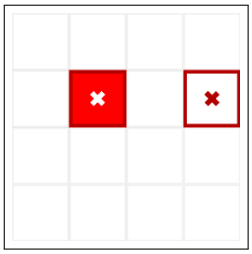


Figure 4.4: An example of the *Mouse* domain. Grids are sizes 4×4 to 9×9 in the test set, and the predicates include *has_goal_box*, *adjacent*, and *adjacent_2*.

An example 4×4 instance is shown in Figure 4.4. Here, the red goal box represents the agent, which can move freely to adjacent cells without obstacles. Given our logical formulation in Equation (4.1), we can rule out insufficient layer size as the cause of poor generalization. In our training set, the maximum Manhattan distance in a 9×9 grid is 18, which is within the representational bounds of our network's layer size $L = 30$. If layer size were the limiting factor, we would expect failures to appear only in grids larger than 15×15 (where the maximum Manhattan distance reaches 30).

Results on *Mouse* In this domain, the model achieves 100% coverage on both the training and in-distribution test sets, with a policy quality of 1.0 (perfect). On the generalization set, the coverage is only 41% with a policy quality of 1.12. This is far from the test set performance and reveals a significant discrepancy.

Figure 4.8 shows the L_1 loss compared to the optimally computed Manhattan distance. As grid size increases, L_1 values grow exponentially, indicating poor generalization of value estimates. We visualized this by generating a problem where the goal is fixed at the center cell, computing the value for each cell, and subtracting the optimal Manhattan distance v^* from the predicted value. Within the training distribution, errors remain small; however, for out-of-distribution grids (e.g., 17×17), errors grow significantly, even for distances well within the training range. Most concerningly, the predicted value for the actual goal state (center cell) deviates by more than 5.0 from the actual value of 0.0. As grid size increases further, the agent becomes effectively unusable, failing even when a single step away from the goal. This phenomenon is shown for 9×9 (in-training) and 17×17 (generalization) in Figure 4.5.

Effect of Grid Size on Goal Prediction We hypothesize that the issue arises because, prior to readout, we sum all embeddings:

$$v(s) = \mathrm{MLP}_{\mathrm{readout}}\left(\sum_{o \in \mathrm{Obj}} x_o^L\right).$$

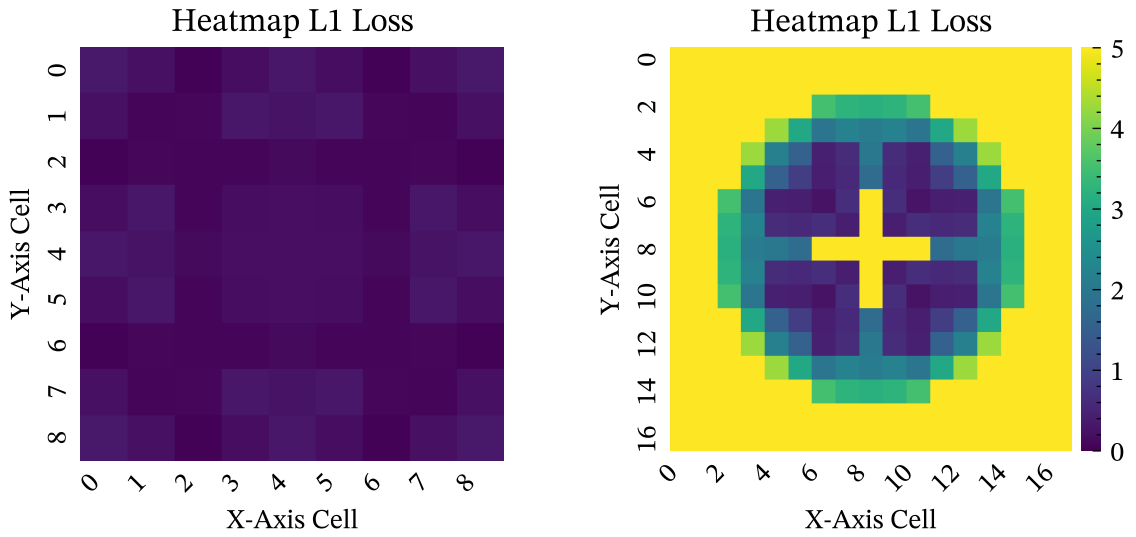


Figure 4.5: Heat maps showcasing the L_1 loss for the default R-GNN model. Each point (x, y) represents the value the network computed if the agent was on the cell, with the goal in the middle. We take the L_1 loss with respect to the real Manhattan distance. On the left is a grid of size 9×9 and on the right 17×17. The symmetries are an artifact of the logical invariances of the positions.

As grid size increases, this sum produces increasingly out-of-distribution inputs for the final readout MLP, which cannot extrapolate effectively. Figure 4.6 plots the average norm of this summation across grid sizes, confirming that larger grids inject more noise into the input.

Intuitively, embeddings from distant and irrelevant objects, such as cells far from the goal, should not have the same influence as those near the agent. This motivates restricting the readout to a subset of objects.

Readout Predicates

One idea is to select only those object embeddings from objects that play a meaningful role in the state. When we increase the number of objects, we usually do not increase the number of meaningful and relevant objects by the same factor. To align with a planning-based perspective, we define a set of *readout predicates*:

$$\text{Readout} \subseteq \text{Preds}.$$

For all atoms $q \in A$ in the current state of the form $q \coloneqq p(o_1, \dots, o_k)$, the readout becomes:

$$v(s) = \text{MLP}_{\text{readout}}\left(\sum_{o \in \text{Obj}_{\text{readout}}} x_o^L\right),$$

$$\text{Obj}_{\text{readout}} = \{o \mid o \in p(o_1, \dots, o_k),\ p \in \text{Readout}\}.$$

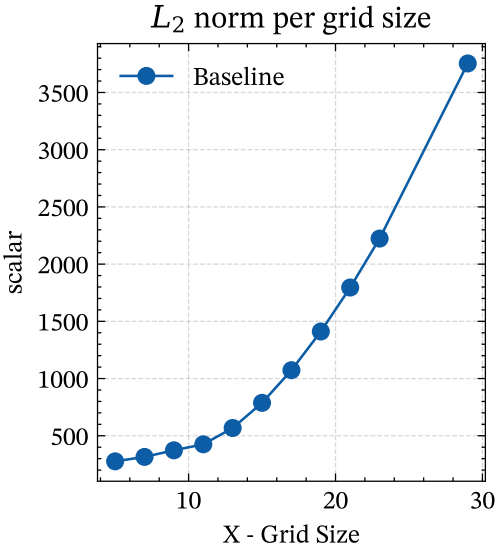


Figure 4.6: Norm of the graph embedding using summation pooling, computed before the readout: norm$(\sum_{o \in \text{Obj}} x_o^L)$. The states are taken from the same "mouse" problem one step before reaching the goal. The only difference between them is the noise introduced by the bigger grids.

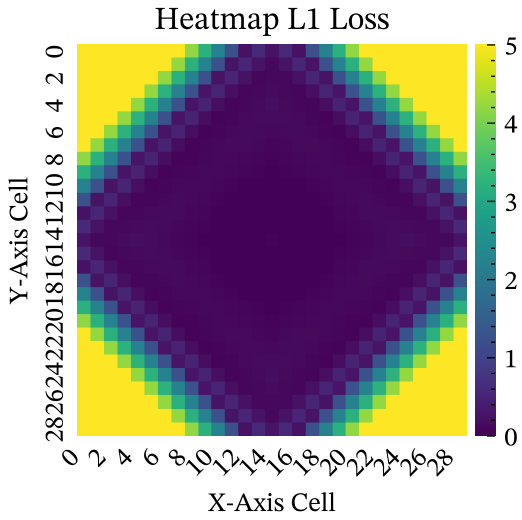


Figure 4.7: Visualization of the attention model's generalization capabilities. A grid of size 29×29 with the goal placed at the center position. The colors indicate how far off the predicted values were at each cell; predictions were made as if the agent was at these positions.

In the *Mouse* domain, we define Readout = {has_goal_box}, meaning the readout occurs only for the goal box (which also represents the agent). This ensures the input to $\text{MLP}_{\text{readout}}$ always corresponds to the agent's cell embedding, bounding the readout norm.

This adjustment dramatically improves results: training achieves 100% test set coverage with policy quality 1.0, and the generalization set achieves 96% with policy quality 1.03.

Furthermore, visualizing embeddings from layer L now yields interpretable patterns: the model has learned to predict the Manhattan distance from any cell to the goal, independent of the agent's position, and the readout at the agent position then extracts the correct value.

Attention-Based Readout

While effective, readout predicates must be manually defined for each domain, introducing human effort and potential bias. Additionally, reading out only at the position of the agent restricts the receptive field to that of the layer size, since effectively all information must be propagated back to the readout predicates. This motivates a more general approach: we employ an attention mechanism, replacing the unweighted sum with a weighted sum.

We split the embeddings in half, using the upper half for attention and the lower part to pass to the readout:

$$x_o^L = \begin{bmatrix} x_o^{\text{up}} \\ x_o^{\text{down}} \end{bmatrix}.$$

$$v(s) = \text{MLP}_{\text{readout}}\left(\sum_{o \in \text{Obj}} \alpha_o x_o^{\text{down}} \right),$$

where $\alpha_o > 0$, $\sum_o \alpha_o = 1$, and attention weights are computed as:

$$\text{score}_o = \frac{x_o^{\text{up}} a^T}{\sqrt{d}} + a_{\text{bias}}, \quad \alpha = \text{softmax}(\text{score}).$$

Here, $a \in \mathbb{R}^d$ is the learned attention vector and $a_{\text{bias}} \in \mathbb{R}$ is a learned bias term. This mechanism allows the model to focus on relevant objects, such as the agent, goal, or obstacles, while keeping the input bounded.

In practice, on the *Mouse* problem, attention achieves identical in-distribution performance (100% coverage, policy quality 1.0) and 96% with a policy quality of 1.03 on the generalization set. Interestingly, the learned attention mirrors the manually defined readout predicate, focusing entirely on *has_goal_box*. The solve rate and policy quality are completely the same as with the readout predicate.

Comparing average L_1 errors in Figure 4.8 shows that both readout predicates and attention significantly improve generalization, while the difference between attention and readout is minimal.

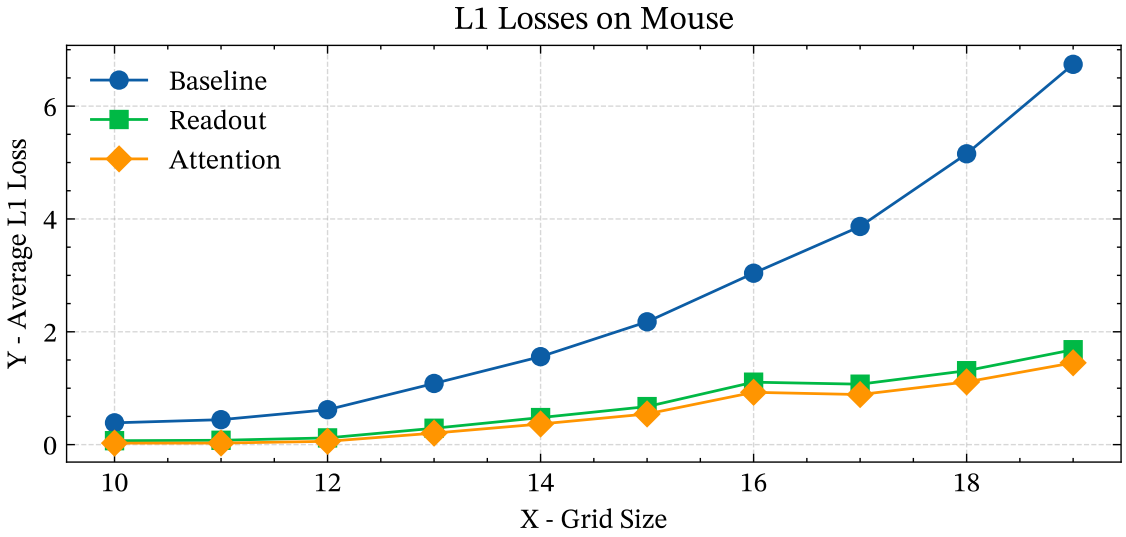


Figure 4.8: Average L_1 loss for the different models: readout with the predicate `has_goal_box` and attention.

Limits of Generalization

Despite these improvements, limitations remain. For example, placing the goal in the center of a large 29×29 grid (as seen in Figure 4.7) reveals that values are accurate only up to a distance of 16; beyond that, values plateau and remain at 16. This likely occurs because the training set (up to 9×9) never contains values above 18. In value-based reinforcement learning, the network may simply fail to extrapolate to unseen higher values.

To test whether layer size is the bottleneck, we trained an attention-based agent with layer size 60 instead of 30. The results were identical, suggesting the problem lies in out-of-distribution value targets, not representational limits. Potential remedies include actor–critic methods or ranking-based value estimation, but we leave these as extensions for future work to remain focused on the current scope.

Nonetheless, our attention and readout approaches already provide substantial improvements in generalization.

4.4 Domains

The design of experimental domains and their corresponding instances is crucial for evaluating the capabilities and limitations of planning models. This section details the fundamental logical building blocks we will use throughout our experiments. We specifically focus on two domains, namely Pushkoban and *PushWorld*.

The motivation for introducing Pushkoban is to study an intermediate problem setting that already exhibits considerable complexity while remaining sufficiently restricted and similar to Sokoban, a domain that is comparatively well understood. The main challenge with *PushWorld* is that it introduces consecutive pushing and objects of varying shapes, which make individual interactions significantly more complex. Pushkoban is a Sokoban-like grid-world setting without walls and with only a single goal object. The challenge in this domain arises from non-goal (normal) boxes, which can also be moved using Sokoban-like dynamics. However, these boxes explicitly do not have associated goal positions; instead, they are designed to serve as obstacles that must be moved aside. This creates medium-difficulty dynamics: we can increase difficulty by adding more boxes, but unlike in Sokoban, adding an additional box does not immediately make the problem combinatorially harder (e.g., doubling the plan length when going from one to two boxes).

4.4.1 Pushkoban

Pushkoban is a domain inspired by the “level 0 puzzles” from *PushWorld* [25], but modified to present a different type of challenge. Instances are generated randomly on complete grids without walls and feature one goal box along with a high density of normal boxes, which occupy

between 20% and 30% of the grid cells. This density of movable obstacles creates puzzle-like scenarios that are complex, yet not as long-horizon and demanding as Sokoban puzzles. We have taken inspiration in the design of Pushkoban from Stante [40].

Fundamental Components and Dynamics

The Pushkoban domain is built upon a set of primitive elements and interaction rules, which we can model logically in PDDL:

- Grid and Cells: The environment is represented as a grid composed of cells, where each cell is a distinct PDDL object. Spatial relationships between cells are defined by the `adjacent(c1, c2)` predicate, which holds if a movable object (such as the player) can move directly between `c1` and `c2`.
- Agent: A single agent (or player) exists in each environment. The agent's location is represented by the unary predicate `has_player(cell)`, which is true for the cell the agent currently occupies. The agent can move between adjacent cells, provided the destination is not occupied by a box or a wall.
- Boxes:
  - Goal Box: A movable object central to the objective. Its position is tracked by `has_goal_box(cell)`. The main objective is to push this box to a designated goal location.
  - Normal Box: A movable obstacle, whose position is represented by `has_normal_box(cell)`. These boxes have no goal location but can be moved, thereby increasing the puzzles complexity.
- Pushkoban Pushing Mechanics: The core interaction involves the agent pushing boxes. A push is possible only if the agent, a box, and an empty cell are aligned in a row. This is enforced with two adjacency predicates:
  1. `adjacent(c1, c2)`: The agent at `c1` is next to the box at `c2`.
  2. `adjacent_2(c1, c3)`: The target cell `c3` is two cells away from the agent's cell `c1` in a straight line.

  A valid push requires:
  `at_agent(c1)`, `has_box(c2)`, `adjacent(c1, c2)`, `adjacent_2(c1, c3)`, and for `c3` to be empty. This restricts interactions to single-box pushes. Unlike in *PushWorld*, chained pushes (e.g., agent → box → box → empty cell) are explicitly forbidden, simplifying the physical dynamics.
- Walls: We do not use walls directly in Pushkoban. Instead, walls are modeled implicitly. Any coordinate without a corresponding cell object acts as a wall. By omitting a cell from the problem definition, no adjacency predicates connect to or from that location, making it impassable.

We generate three distinct sets of problem instances to comprehensively evaluate model performance:

- Training Sets: 2000 instances are generated on square grids ranging from 4x4 to 7x7, with 500 problems per size.
- Test Set: 200 instances generated using the same configuration and distribution as the training set. This set evaluates in-distribution performance and helps assess overfitting (50 per size).
- Generalization Set: The difficulty is scaled by increasing grid sizes from 8x8 up to 15x15, with 25 instances per size (200 in total). Larger grids indirectly increase the number of obstacle boxes, since density is tied to grid size. These instances test generalization by evaluating whether models can transfer learned knowledge to more complex, unseen problems.

Three example instances from the Pushkoban domain are shown in Figure 4.9.

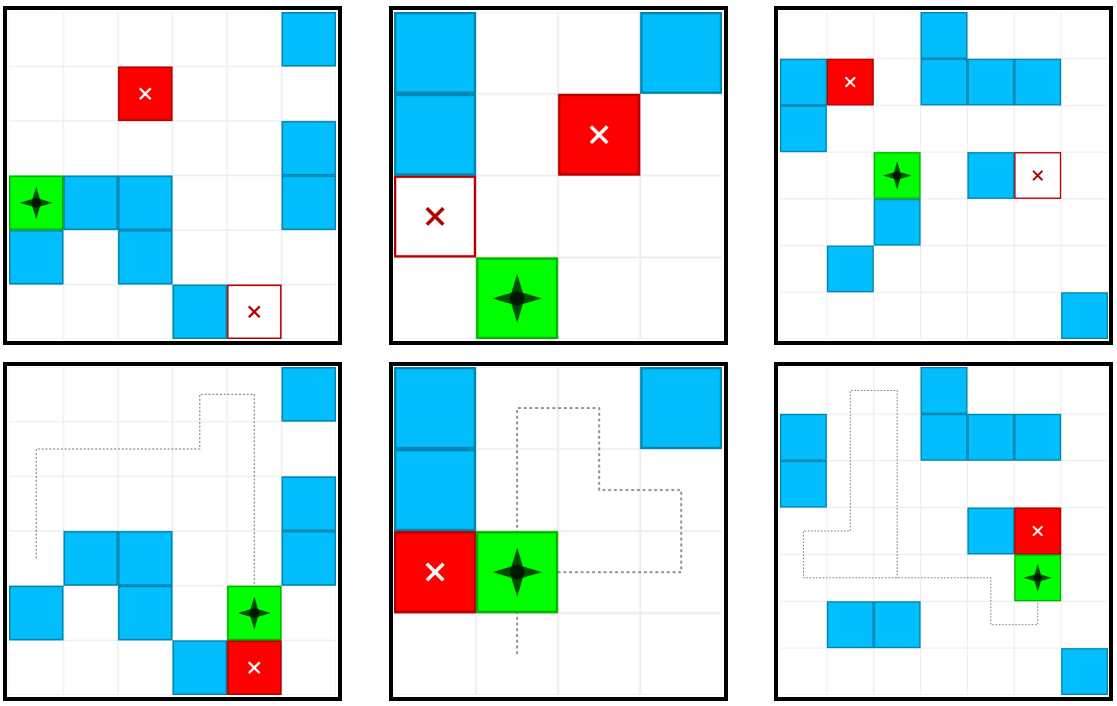


Figure 4.9: Examples of three randomly generated Pushkoban problems; their initial state (up); and an example solution (below). Each puzzle requires the red outlined box (with the red cross in the middle) to be covered by red block (white corss in the middle). The player may move the green agent (as a whole unit) up, down, left, or right. Other pushable objects (blue boxes) are in the way and sometimes have to be moved in order to reach the goal.

Problem Set Generation

For the Pushkoban domain we adopt a simple rejection sampling strategy: problems are generated fully at random by placing blocks, goals, and the player at random positions. Using a solver (in our case LAMA), we then attempt to solve each problem with a timeout of 100s, which should be enough given the small instance sizes. We generate approximately twice as many problems as needed, then filter out unsolvable instances. From the remaining problems, we further remove those with the lowest plan lengths to introduce a slight bias toward more complex tasks. This prevents trivial instances requiring only a few steps. All problems are available on GitHub[1] in both PDDL and text-based formats, enabling alternative encodings (e.g., images or other logical representations).

4.4.2 PushWorld

The *PushWorld* benchmark [25] was released in four levels. Of particular interest for our work is the level 0 suite, which was designed for learning-based planning methods. The level 0 suite itself is divided into the following categories: *base*, *size*, *walls*, *obstacles*, *shapes*, *goals*, and finally *all*.

Each category consists of 2000 training problems and 200 test problems, sampled from the same generation process. The problems are generated procedurally using rejection sampling and can be extended, but for fairness we use the exact same instances employed by the original authors [25] when training their deep learning agents.

The categories are defined as follows:

- Base: Grids of size 5x5, three walls, one goal box, and one normal box. All shapes are 1×1.
- Size: Grids ranging from 5x5 up to 10x10 (not necessarily square; e.g., 5x7). Otherwise identical to *base*.
- Walls: Same as *base*, but with 3–5 walls (selected randomly).
- Obstacles: Same as *base*, but with two normal boxes.
- Shapes: Includes differently shaped polyominoes for all objects with sizes of 1, 2, or 3 (chosen randomly).
- Goals: Same as *base*, but with two goal boxes, one of shape 1×2 and the other 1×1.
- All: Grids of size 5x5 up to 10x10, 3–5 walls, 1 or 2 normal boxes, 1 or 2 goals, and random shapes of size 1–3. An additional constraint ensures that two goals do not share the same shape.

In addition to providing problems in text and image format, *PushWorld* also includes a PDDL parser that specifies the problem domain and instances in PDDL format.

[1] `https://github.com/y-hesse/pushkoban-pddl-levels`

The PushWorld PDDL

The domain representation in *PushWorld* uses the same core concept of connected cell objects to represent the grid, but encodes obstacles and goal boxes differently. Each box and the player are modeled as distinct PDDL objects, linked to grid positions with the predicate at(box, pos), which specifies that an object is located at a particular grid cell.

To capture movement, four direction objects are introduced: $(l)eft$, $(r)ight$, $(u)p$, and $(d)own$. Two cells are connected by a ternary predicate, e.g., adj(cell_0_0, cell_1_0, r). This avoids the need to explicitly model the adj_2 relation we introduced in Pushkoban. However, this representation introduces additional objects and may make reasoning about connectivity less straightforward. We revisit this issue in our section on expressivity.

To handle consecutive object pushing, *PushWorld* uses a two-action approach:

1. Move action: Specifies the direction of movement. This action sets the predicate should-move(agent, dir) for the player/agent object. Importantly, move actions can only be applied if no object in the instance already has a should-move predicate.
2. Push action: Executes the push. It moves the marked object one step in the specified direction. If another object blocks the destination, that object is also marked with should-move. If any marked object cannot move (e.g., because of a wall or grid boundary), no further actions are possible and the state becomes a dead end.

This differs from most planning problems, where applicability is precomputed before executing an action. Here, a move in any direction can be proposed, but the resulting dynamics unfold only when consecutive push actions are applied until all should-move predicates are resolved.

Object interactions are further handled using a precomputed predicate in-collision:

```
(in-collision
    ?obj - moveable-object
    ?pos - position
    ?other-obj - moveable-object
    ?other-pos - position
)
```

This predicate is provided as a static atom in each PDDL problem instance. For every pair of objects (o_i, o_j) it specifies whether placing them at positions (x_0, x_1) would yield an illegal state. The predicate is critical for puzzles involving polyomino-shaped objects: it abstracts away shape details and allows legality checks to be reduced to table lookups.

Each box object is placed at a single grid cell (its "head"), while the rest of its shape is implicitly defined via in-collision. However, this representation is extremely memory intensive: for grids of size n with m objects, the predicate introduces $\mathcal{O}(n^2 \cdot m^2)$ atoms. While manageable in the

level 0 suite (5x5–10x10 grids with few obstacles), this becomes prohibitive in larger domains (level 1 and beyond). In fact, training our symbolic neural network becomes infeasible due to the exponential atom blow-up, even on GPUs with more than 48 GB of VRAM. Classical planners, on the other hand, benefit from this precomputation, as it reduces the runtime cost of validating actions.

Adapted PushWorld PDDL

To address the scalability issues caused by the in-collision predicate, we designed a simplified encoding. The goal is to preserve the core dynamics of the domain while mitigating the exponential blow-up of atoms.

In this adapted encoding, we remove the in-collision predicate and eliminate the notion of "head" positions for movable objects. Instead, each object is explicitly connected to all the cells it occupies, for example: (at obj_0 cell_0_0), (at obj_0 cell_1_0), ..., (at obj_0 cell_5_0).

We retain the explicit modeling of directions. In addition, we introduce a new action, push-done, alongside the original move and push actions. The semantics are as follows: triggering a move marks an object with the predicate should-move. Unlike in the original formulation, however, a push does not move the entire object at once. This adjustment is made primarily for performance reasons during action grounding. If the entire shaped object were pushed in a single step, the grounder would need to check all positions the object occupies, and for each of these positions verify whether a move to the neighboring cell is possible. This process results in quadratic scaling with the grid size.

To avoid this, each push moves only a single cell from one position to the next. Once a cell has moved, it is marked with the predicate (has-moved ?obj ?pos). After all cells of an object are marked as moved, the should-move predicate can be cleared using the new push-done action. This approach significantly improves performance, especially for larger problem instances such as those in level 1 and above.

However, this formulation also introduces a drawback: there are now multiple possible sequences of push and push-done actions leading from a state where the move is possible to the next, whereas in the original encoding only a single push sequence was possible. This unnecessarily complicates planning for both classical planners and learning-based models. Importantly, regardless of which path is followed, the system eventually reaches the same resulting state where the next move action becomes applicable.

To simplify training and evaluation, we therefore abstract away intermediate push and push-done actions. Concretely, if the executed sequence is move_1, push, ..., push-done, move_2, the agent only observes the states before move_1 and move_2, as these are the states where a decision matters.

This design choice reduces computational overhead and better aligns with reinforcement learning baselines (e.g., PPO, DQN) as presented in [25], where each action immediately yields the resulting state after all push dynamics have been resolved.

The full PDDL specification of the adapted *PushWorld* encoding is provided in the appendix (Section A.1).

4.5 Expressivity

We have already seen that GNNs are, in general, limited in their logical expressive power somewhere between $\mathcal{GC}_2$ and $\mathcal{C}_3$ [4, 39]. This is a significant concern when using GNNs. The expressivity argument essentially boils down to the fact that, given two planning states s_1, s_2, the R-GNN cannot differentiate between them. As such, if $v^*(s_1) \neq v^*(s_2)$, the R-GNN would still predict $v^{gnn}(s_1) = v^{gnn}(s_2)$.

4.5.1 Intuition of $\mathcal{C}_2$

Within the $\mathcal{C}_2$ fragment of first-order logic, we have already seen in Chapter 2 that simple path-finding problems can be solved. But what about paths that concern multiple objects? For example, in the Pushkoban domain we encounter exactly this issue. We have the player and the goal box; in order to move the goal box, we must reason logically over multiple objects.

Let us argue why we can still solve this, as it provides some interesting insights into the expressivity within $\mathcal{C}_2$ logic. We are in the simplified Pushkoban domain, where we have the following predicates: E, E_2, G (goal), B (goal box), and P (player). We assume a perfect grid without obstacles (i.e., no other movable boxes). How can we solve this problem? Figure Figure 4.10 shows a solution sketch of the problem under consideration. The main concepts we need to consider are moving the player towards the box, aligning the box with the goal along one axis, repositioning the player to enable pushing towards the goal, and finally, pushing the box into the goal.

One of the main questions is: how do we test if two positions share an axis? Once this is determined, the rest of the problem becomes quite simple. The idea is to use the fact that if two objects are not on the same axis, there exist exactly two shortest paths connecting them, whereas if they are axis-aligned, only a single shortest path exists. Additionally, we abstract away from positions directly and instead talk about the unary (in this case unique) predicates that hold at their respective positions.

If we want to test, for example, whether two predicates are aligned and distance l apart, we can use the following logical formula:

$$aligned_l(pred1, pred2) = \exists x\ pred1(x) \wedge \exists^{=1} y\ E(x, y)\ sp_{l-1}(y, pred2)$$

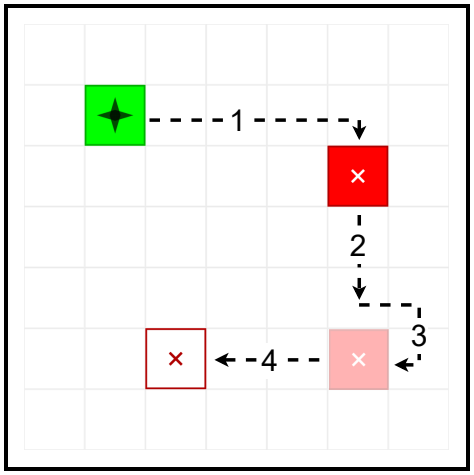


Figure 4.10: An example of how to logically deconstruct the agent's plan in the simplest setting of a single goal box (red box), an agent (green), and a goal (red cross), in a complete grid.

Given this formulation, we can solve the rest of the problem:

$$
\begin{aligned}
p_0(x, pred) &= pred(x) \\
p_i(x, pred) &= \exists y\ E(x,y) \wedge p_{i-1}(y, pred) \\
sp_i(x, pred) &= p_i(x, pred) \wedge \neg p_{i-1}(x, pred) \\
push\_from_i(x) &= \exists y\ B(y) \wedge \mathrm{sp}_i(y, G) \wedge \mathrm{sp}_{i+1}(x, G) \wedge \mathrm{E}(x, B) \\
V^*(s) &= \min_{i,j} \left((i+j) \cdot [\![\exists x\ push\_from_i(x) \wedge \mathrm{sp}_j(x, P)]\!] \right) \\
&\quad + 2 \cdot \sum_k [\![\exists x\ B(x) \wedge \neg aligned_k(x)]\!]
\end{aligned}
$$

In this formula we compute two distances: the distance from the player to the position from where the box can be pushed (j), and the distance between the box and the goal (i). If the box and the goal are not aligned, the player must first push the box into alignment before pushing it towards the goal again. This additional maneuver takes two extra steps, so we add this to the final value by checking whether the box and goal are aligned. All of the above logical formulas are expressed in $\mathcal{GC}_2$, and therefore this value function can be computed using an R-GNN.

4.5.2 $\mathcal{C}_2$ and Pushkoban

One way to determine whether we are actually facing R-GNN expressivity issues is to test this explicitly. We can encode two different states as their respective graph representations and then run the 1-WL coloring algorithm to determine whether these two graphs are equal under 1-WL. Doing this for all pairs of states in our environment would yield a clear result regarding the expressivness of GNNs on that problem set.

This is exactly what Drexler et al. [11] did in their work on "Symmetries and expressive requirements for learning general policies." They provide code that allows us to test these pairwise 1-WL equivalences. Their graph structure differs slightly from ours, but the results should be identical as the underlying encodings are conceptually very similar. Running their code on our Pushkoban 4×4 training problems yields the following result:

Domain	#Q	#S	1-WL Collisions	1-WL Collisions and diff v^*
Pushkoban 4x4	500	3.5M	0	0

Here we clearly see that not only are there no collisions where two different states would be assigned the same value function by the GNN, but there are no collisions at all.

It is important to emphasize, however, that this does not mean we face no expressivity issues in this domain. First, the size of these input problems was limited to 4×4 due to the enormous state space of larger grids. This restricted the number of state pairs we could analyze.

Additionally consider the following example: given two states that cannot be distinguished under 1-WL, suppose we add a box in the corner that does not affect the optimal value function. This would already be enough for the two states, and thus the two graphs, to become distinguishable under 1-WL and therefore by our R-GNNs. In this way, the R-GNN may not have the expressivity to truly solve the problem but, given this additional box, it can overfit on the training set and in turn solve the trainings-problem. This does not help with generalization; in fact, it hinders the discovery of a general policy, as the irrelevant box should not affect the logic of the state at all.

Thus, the takeaway is: if the R-GNN can solve all problems on the training set, it may be able to differentiate all states, but that does not mean it has captured the domain logic. Moreover, if there is a significant drop from training to test performance, we can probably assume that the agent has, at least partially, overfitted to irrelevant features in the training set and we might have an issue with the expressive power of the R-GNN in this domain encoding.

4.5.3 Derived Predicates

One approach that has been shown to enhance the logical power of R-GNNs [37] is the use of *derived predicates*. Derived predicates are special predicates that can be defined in a PDDL domain description. They allow us to define a predicate that holds whenever a predefined logical formula holds.

For example, one could define:

```
1 (:derived (pushable ?x ?y)
2     (and
3         (adjacent ?x ?y) (has_box ?x) (not (has_box ?y))
4     )
```

```
)
```

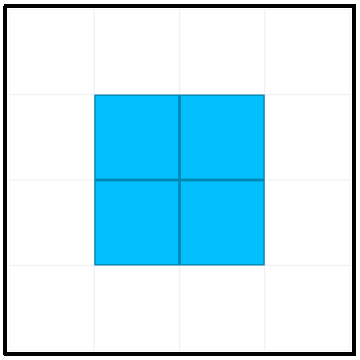


Figure 4.11: An example of a quad structure of boxes in Pushkoban. None of the boxes can ever be moved again once they are in such a construction.

This derived predicate indicates whether a box is next to an empty cell. In this case, it is easily expressible in $\mathcal{C}_2$. However, derived predicates can also be used to overcome expressivity limitations in cases where the logical formula cannot be represented in $\mathcal{C}_2$ but can still be expressed logically.

For example, detecting an immovable block (Figure 4.11) requires reasoning over diagonally attached blocks. This is not easily achievable with only two variables. Within $\mathcal{C}_4$, however, this is a straightforward formula:

$$\begin{aligned} \mathit{blocked}(x_1, x_2, x_3, x_4) = \mathit{box}(x_1) \wedge \mathit{box}(x_2) \wedge \mathit{box}(x_3) \wedge \mathit{box}(x_4) \\ \wedge E(x_1, x_2) \wedge E(x_2, x_3) \wedge E(x_3, x_4) \wedge E(x_4, x_1). \end{aligned}$$

It is possible to approximate this in $\mathcal{C}_2$, but there is no simple way to check locally whether a block is blocked without explicitly reasoning over its surrounding neighbors.

Suppose we were given the following derived predicate that represents four objects connected to each other as seen in Figure 4.11:

```
(:derived (quad ?a ?b ?c ?d)
    (and
        (adjacent ?a ?b)
        (adjacent ?b ?c)
        (adjacent ?c ?d)
        (adjacent ?d ?a)
        (not (adjacent ?a ?c))
        (not (adjacent ?b ?d))
    )
)
```

Given this derived predicate it is not straightforward how to translate this into a $\mathcal{C}_2$ formula. For instance,

$$blocked(x_1, x_2, x_3, x_4) = box(x_1) \wedge box(x_2) \wedge box(x_3) \wedge box(x_4) \wedge quad(x_1, x_2, x_3, x_4)$$

still requires reasoning over four variables.

Here we can employ a useful trick: $quad(x_1, x_2, x_3, x_4)$ is an atom and, as such, is represented implicitly as a node in the hypergraph defined by our R-GNN encoding. Thus, we can argue over this atom node $a \coloneqq quad(x_1, x_2, x_3, x_4)$. This allows us to define:

$$blocked(x) \coloneqq \exists a\ quad(a) \wedge E(x, a) \wedge \exists^{=4} x\ box(x) \wedge E(x, a).$$

In this case, E refers to the graph connection and is no longer tied directly to instance-level adjacency.

With this trick, even derived predicates with more than two arguments can help enhance model expressivity.

Another interesting derived predicate is `push`. In general, the push action can be expressed as:

$$push(x, y, z) = E(x, y) \wedge E(y, z) \wedge E_2(x, z),$$

omitting the conditions that there is a box on y, the agent is on position x, and that z is free (for simplicity). If the R-GNN wants to plan, it would be beneficial if it could simulate these actions on the graph structure to construct plans. However, this *push* is a logical formula in $\mathcal{C}_3$. Of course, approximations exist, and experimentally we can see that the agent can learn to navigate the Pushkoban world without this dreived predicate. Logically, however, such a *push* predicate would be very useful. For this reason, we explicitly test this predicate as one of the additional derived predicates:

```
(:derived (push ?a ?b ?c)
    (and
        (adjacent ?a ?b)
        (adjacent ?b ?c)
        (adjacent2 ?a ?c)
    )
)
```

4.5.4 Directions

One interesting issue from a logical standpoint is the encoding of directions. In our Pushkoban domain, we omit explicitly modeling directions, as the structure is already well defined through

the two adjacency relations (E and E_2). In the *PushWorld* domain, by contrast, explicit objects are used to define directions.

Another possible approach would be to use four unique adjacency relations, e.g., E_l, E_r, E_u, E_d. While this would work, it would destroy the invariances that are central to our approach. A flipped problem instance would no longer be logically equivalent to the original one. This invariance is crucial and gives us an advantage over CNN approaches. Thus, it is quite elegant to model directions using objects, since their names have no impact on logical representation, preserving invariance under rotations and flips.

Since these direction objects are constants, as long as we can distinguish them, we can write formulas that include them. Implicitly, we can still write $\mathcal{C}_2$ formulas using $E_l, \ldots, E_d$. For example, encoding a blocked structure is straightforward:

$$blocked(x) = box(x) \wedge \exists y\ box(y) \wedge E_r(x, y) \wedge \exists x\ box(x) \wedge E_d(y, x) \wedge \exists y\ box(y) \wedge E_l(x, y)$$

However, now we require additional formulas to make the representation invariant under rotation. In fact, we need four such formulas to cover all possible starting positions. In theory, this object-direction encoding increases the power of our R-GNN, but it also increases computation.

Furthermore, this approach only works if we can differentiate and reason over l, r, u, d. Grouping them into horizontal and vertical directions is straightforward, but separating up from down is significantly harder and may not be logically possible in certain states. For this reason, it is not entirely clear whether this encoding of directions is superior or inferior to the two adjacency encoding, which motivates an experimental evaluation.

5 Results

In this section, we present the experimental results of our work. First, we describe how the experiments were constructed and conducted, including the hyperparameters used. Next, we demonstrate that using AV* effectively addresses the sparse reward problem that traditional reinforcement learning methods struggle with. To this end, we compare our AV* approach to the policy gradient reinforcement learning approach used in Ståhlberg et al. [38], both using the same R-GNN-based logical encoding backbone. In a second experiment, we evaluate AV*'s performance on Pushkoban and investigate the factors that contribute to successful learning in this domain. Here, we perform an ablation study on the additional improvements introduced in Chapter 4. We also examine the performance of AV* when combined with a CNN backbone and discuss our observations. In a third experiment, we test AV* on *PushWorld* and specifically compare the results against PPO and DQN [25], which serve as the benchmarks baselines. Finally, we show that our model trained on level 0 is competitive with the classical planner LAMA when evaluated on level 1 problems.

5.1 Experimental Setup and Hyperparameters

We outline our experimental setup and hyperparameters in this section. All experiments were conducted on compute nodes equipped with NVIDIA L40S GPUs and Intel(R) Xeon(R) Gold 6542Y processors. Each run used a single GPU and 8 additional CPU cores running for 48 hours. The computational resources were provided by the Chair of Machine Learning and Reasoning (i6) of RWTH Aachen[1].

Each experiment was run twice and we report results from the model that achieved the higher performance on the test set at the end of training. Variance across runs was minimal, largely due to the training methodology (learning a state-value function directly rather than using policy gradient methods) as well as the long training durations. Since our focus lies on generalization, the test set is best interpreted as a proxy for validation performance.

Our default hyperparameters are a layer size of $L = 30$ and an embedding dimension of 32. We employ the Adam optimizer [26] with a learning rate of 0.0001 for the GNN and 0.001 for the readout module. These values were chosen empirically during method development and proved robust across experiments. During training, we restricted the search to a maximum of 2048 evaluated nodes. We use a dead end penalty of -200; this is also the bound on our maximal rollout length. The buffer size includes 40 batches each with 128 state-target pairs. To

[1] `https://ml.rwth-aachen.de/`

generate training data four parallel workers are used; each refreshing their model parameters after providing 20 batches.

In A* search, state expansions is prioritized for a state s using

$$f(s) = 0.5\, g(s) + v^{gnn}(s) + 1 - 0.5t,$$

where $g(s)$ denotes the number of steps from the initial state and t is a random variable uniformly sampled from $[0, 1]$. Assigning a weight of 0.5 to the step size encourages the search to follow the heuristic more greedily. Since the approximated value function is not neccesserily admissible finding the shortest solution path is not a focus and as such we do not require a weight of 1. Empirically, this choice performed well. The random term introduces variance in node exploration, analogous to exploration during execution. Additionally, with probability $\varepsilon = 0.05$, a newly generated state is placed at the front of the queue. This modification both preserves the theoretical properties of the algorithm and promotes further exploration. Policy rollouts are terminated if they exceed 200 steps. An empirical cutoff, as all problems in Pushkoban and *PushWorld* level 0 can be solved within that bound. A more extensive hyperparameter search was not conducted due to the high computational cost of training.

5.1.1 Evaluation Metrics

We evaluate models primarily in terms of the *solve rate*, defined as the ratio of solved problems to the total number of problems. We also measure *plan quality*, defined as the ratio of the number of steps taken by our model to the number of steps LAMA required for solved instances, averaged across all solved problems. A plan quality of 1.0 or below corresponds to a solution at least as good as LAMA's. Fast Downward is used with the option "seq-sat-lama-2011" (LAMA) [32] [2].

We consider two evaluation principles:

1. Agentic greedy approach. The model acts as an agent in an environment, selecting actions greedily based on estimated value. This corresponds to a reinforcement learning perspective: each selected action is executed, and failure occurs once no further progress is possible.

2. Search-guided approach. Here, the value function acts as a heuristic for an A* search. Multiple paths may be expanded in parallel, and the search can switch between them depending on the most promising state. This approach leverages domain knowledge. Performance is measured in terms of the number of expanded states, the time required to find a solution, and the resulting plan length.

One challenge of the greedy agentic approach is cycling: the agent may oscillate between states when trapped in a local maximum ($v(s) > v(s') \;\; \forall s' \in N(s)$). This behavior can occur

[2] `https://www.fast-downward.org/latest/documentation/ipc-planners/`

if learning has not fully converged or if noise is introduced by multiple workers updating similar states simultaneously. To mitigate this, we introduce stochasticity in action selection. Specifically, let

$$m = \max(v(N(s)))$$

and

$$\text{next} = \text{uniform\_choice}[s' \mid |m - v(s')| < 0.5,\ s' \in N(s)].$$

This ensures that if multiple successor states are nearly equally promising, one is chosen at random, which helps prevent cycling. Another common remedy is the so-called *closed stochastic* setting, where revisiting previously visited states is explicitly forbidden, but we choose not use this as the above formulation allows the agent to backtrack in case a chosen action is wrong.

5.2 Comparison with Model-Free Actor-Critic RL

We compare our value-based R-GNN trained with AV* against an actor-critic agent trained with a model-free reinforcement learning approach that uses a replay buffer and full rollouts. The dataset and comparative results were taken from Stante [40], who employed the same R-GNN but in a purely model-free policy gradient setup, akin to Ståhlberg et al. [38]. To ensure comparative results, we trained on the same datasets as Stante (V1, V2, and V3). V1 is similar to Pushkoban but without any obstacle boxes and with one agent and one box in a square grid. V2 concists of Sokoban problems with a single goal box and many walls. V3 is equivalent to our Pushkoban environment, but with a different number of obstacles.

Training instances used grid sizes from 4×4 to 10×10, while test instances extended up to 15×15. Stante reported results with and without a dead-end heuristic; for simplicity, we only include their results without heuristics. They also employed a closed stochastic sampling scheme for state transitions. Results are summarized in Table 5.1.

	AC RL		AV* + R-GNN	
	Train (%)	Test (%)	Train (%)	Test (%)
V1	100.0 (70/70)	100.0(60/60)	100.0 (70/70)	96.7 (58/60)
V2	69.7 (244/350)	54.2 (65/120)	99.4 (348/350)	84.2 (101/120)
V3	51.1 (179/350)	69.2 (83/120)	97.1 (340/350)	78.3 (94/120)

Table 5.1: Comparison of the actor-critic reinforcement learning (AC RL) agent and our value-based AV* + R-GNN agent. Train/Test (%) indicate coverage on this problem set, with the number of solved cases in parentheses. Higher is better.

Discussion. The results clearly highlight advantages of the AV* algorithm. In training, AV* nearly solved the entire dataset, suggesting that the training data was fully exhausted and that additional instances could further improve performance. On the test set, performance dropped

by up to 20% points compared to the training performance but still consistently outperformed the actor-critic baseline in all environments except for V1.

We attribute this training–test gap primarily to the test set design of [40]: instances extended up to 15×15, whereas training was limited to 10×10. The two unsolved cases in the V1 test set are of size 15×15, indicating a generalization issue, as discussed in Section 4.3. This suggests that extensions such as the attention-based readout could improve generalization further. Interestingly, the AC RL approach did not struggle with generalization in V1, solving all test cases with 100% accuracy.

In the work of Ståhlberg et al. [36, 38], policy gradient methods generally outperformed purely value-based models. This makes our results particularly impressive, as we clearly outperform a policy gradient approach using only a value-based method. Overall, these findings support our hypothesis that in environments with sparse rewards, a structured, search-informed training approach provides substantial advantages over purely model-free reinforcement learning.

5.3 Pushkoban

In this section, we focus specifically on the Pushkoban problem set and present an ablation study to evaluate which components contributed to performance and which did not. The section is organized as follows: we first describe the ablation study and summarize the key takeaways from the results. The subsequent subsections analyze specific aspects of the approach, including generalization, expressivity, and the progressive training.

5.3.1 Pushkoban Results

We use the Pushkoban domain to evaluate how well our method serves as a proxy for the more complex *PushWorld* environment. In Chapter 4, we explored various potential issues and improvements, both in the model architecture and in the logical encoding of the domains. These aspects include progressive training to reduce dependence on a fixed layer size L; generalization challenges, as experimentally observed in Section 4.3; the addition of derived predicates to enhance the logical expressivity of the R-GNN (e.g., *push*, *quad*); and an alternative domain encoding that uses objects as directions directly, rather than relying on adj_2.

Each of these factors could impact training, testing, and generalization coverage, motivating an ablation study. All models are trained on the same Pushkoban problems, varying only in the use of attention readout, derived predicates, and directional encoding. Training was performed using AV* with default hyperparameters. For each experiment, we selected the run achieving the highest test coverage after 48 hours.

For comparison, we also ran AV* using a CNN backbone, described thoroughly in Section A.2. Results include both coverage and plan quality. For derived predicates, we added *push* and

quad. Progressive training was evaluated only on the default Pushkoban encoding to provide a direct comparison to the baseline, as this ablation setting is not ideal for training with a progressive loss (the grids are at most 15×15, within the bounds of message passing). The progressive loss experiments were conducted with a fixed layer size of $L = 30$ at test time, with further analysis in a later subsection.

Experiment Type					Train		Test		Generalization	
Misc	Attention	Push	Quad	Direction	Cov% ↑	PQ ↓	Cov% ↑	PQ ↓	Cov% ↑	PQ ↓
	✗	✗	✗	✗	96.8	1.019	89.5	1.0055	68.5	0.988
	✓	✗	✗	✗	98.3	1.011	88.5	1.048	83.5	0.955
	✗	✓	✗	✗	99.0	1.005	95.5	1.033	76.5	1.021
	✓	✓	✗	✗	98.4	1.017	94.5	1.052	82.0	1.008
	✗	✗	✓	✗	97.4	1.011	89.5	1.038	62.5	0.972
	✓	✗	✓	✗	97.9	1.007	90.0	1.037	84.0	1.052
	✗	✓	✓	✗	98.7	1.005	94.0	1.027	80.0	1.079
	✓	✓	✓	✗	99.1	1.006	95.5	1.020	80.5	1.055
	✗	✗	✗	✓	97.4	1.014	85.5	1.047	53.0	1.035
	✓	✗	✗	✓	97.7	1.012	88.5	1.049	69.5	1.030
Prog. Training	✗	✗	✗	✗	98.0	1.009	90.5	1.036	74.5	0.965
Prog. Training	✓	✗	✗	✗	98.1	1.011	91.5	1.041	75.6	1.053
CNN					90.5	1.589	83.0	1.718	47.5	2.776

Table 5.2: Comparison of models across Train, Test, and Generalization on Pushkoban. Trained with AV*. Cov% indicates coverage percentage; PQ indicates performance quality. Higher values (↑) are better. Checkmarks indicate experiment type features. With additional experiments including "progressive training" and "CNN" as a comparison against our R-GNN.

Discussion. The Pushkoban results are summarized in Table 5.2. All R-GNN models perform well on the training set, achieving at least 95.0% coverage; even the CNN reaches 90.5%. This confirms that the models successfully learned the training instances. Policy quality is consistently low (below 1.1), except for the CNN, indicating that found solutions are near-optimal. Test coverage remains high, above 85.5% for the R-GNN models, suggesting minimal overfitting to problem-specific features. Some R-GNN models achieve up to 99.1% coverage on the training set, but performance on the test set is consistently lower, with a gap of about 5%-10%. This suggests that additional training problems might have improved generalization; nevertheless, policy quality on the test set remains above 1.1, indicating that solutions are still near-optimal.

This trend continues in the generalization set, where policy quality is again low, occasionally below that of LAMA, which considering the larger grids of 15×15 problems is expected to result in suboptimal LAMA plans. Nevertheless, all R-GNN models achieve at least 69.5% coverage when using attention, demonstrating strong generalization.

The CNN exhibits the weakest generalization performance among the evaluated models. While its training and test outcomes remain competitive, the resulting policy quality is comparatively

higher, suggesting suboptimality. This limitation may stem from the absence of an explicit logical representation. Additionally, in each forward pass a new image is produced via data augmentation, a mechanism that may facilitate escape from local loops but simultaneously increases plan lengths. Most notably, the CNN shows the largest disparity between test and generalization coverage, with a drop of 35.5%, thereby emphasizing the superior generalization capacity of R-GNNs.

5.3.2 Generalization

The results in Table 5.2 show very promising generalization performance, far above that of the CNN. The worst R-GNN model using attention still achieves 69.5%, which is 23.5% better than the CNN. Focusing on the effect of using the attention readout introduced in Section 4.3, we find strong evidence that attention almost always improves generalization. For example, in the default case, adding attention boosts generalization coverage from 68.5% to 83.5% (an improvement of 15% points) with almost no impact on test performance. This outcome aligns with our expectations, as the attention readout does not enhance, nor limit, the expressive power of the underlying GNN but rather refines the aggregation of information. The benefit of attention holds across nearly all experiments, with two notable exceptions. First, in the setting with both derived predicates (*push* and *quad*), generalization coverage is only slightly increased when attention is used. We discuss this curious outlier further in Section 5.3.3. Still, even without attention, this model achieves strong generalization. Second, attention does not really help in combination with progressive training. The largest gain from adding attention occurs in the directional encoding experiment, where coverage increases from 53.5% to 69.5%. However, even with this improvement, the model only slightly outperforms the default *adj* and adj_2 encoding without attention (68.5%). This result is surprising, since training and test coverage remain comparable to the default case, yet generalization lags behind.

An inspection of the attention patterns reveals further insights. In both the default and *push* models, nearly all attention is focused on the goal position. The *quad* model, by contrast, assigns attention primarily to the cell occupied by the player. In the combined *push+quad* model, the attention mechanism exhibits a notable sub-goal-like structure: the model focuses on the cell from which the goal object should be pushed next. This results in an interpretable pattern, as illustrated in Figure 5.1. Interestingly, this sub-goal structure does not seem to improve performance. In fact, generalization coverage only increases slightly from 80.0% without attention to 80.5% with attention, suggesting that this way of assigning attention may not be optimal.

For the progressive loss model, attention tends to be more diffuse, often centered around the goal and occasionally concentrated entirely on the goal cell. In the directional encoding experiment, attention is again placed exclusively on the goal box. Overall, attention leads to major improvements in generalization across most models. However, some combinations

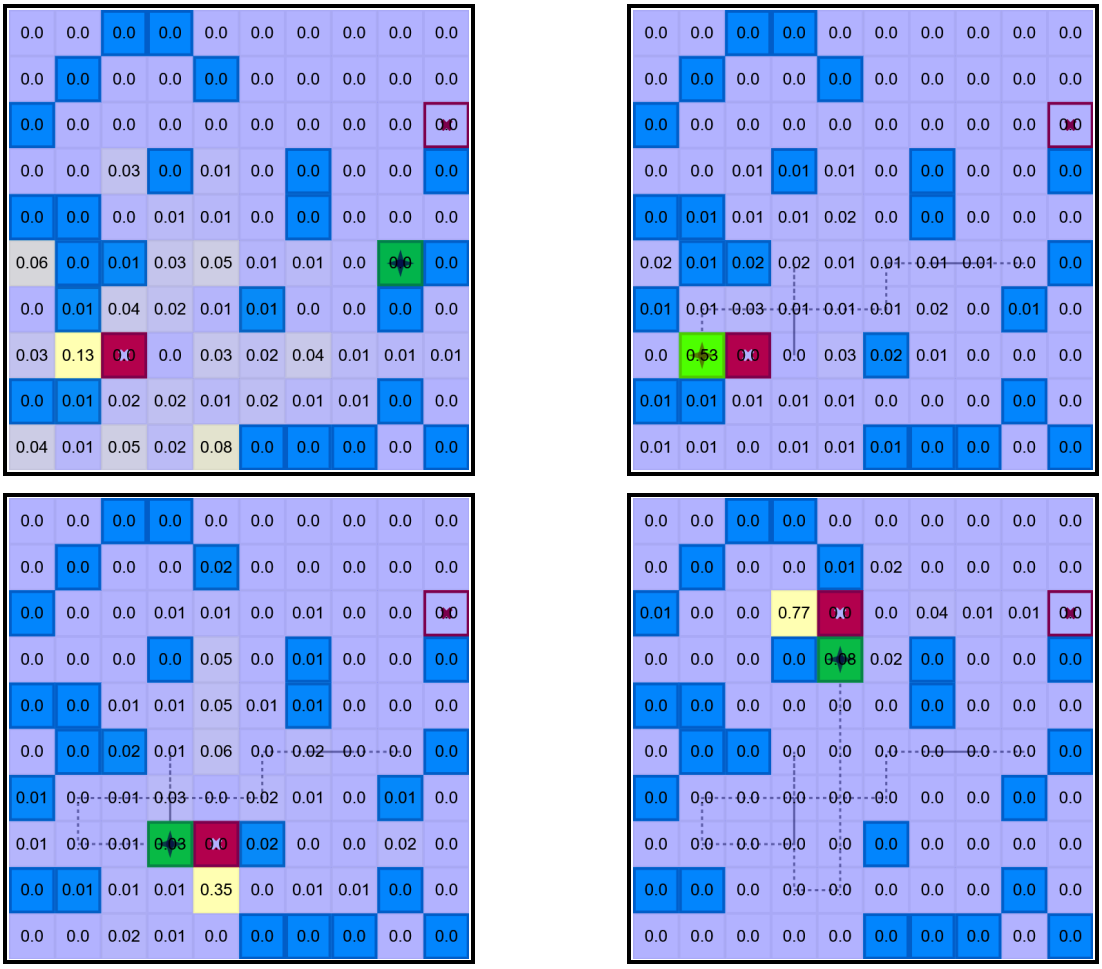


Figure 5.1: Visualization of attention readout in Pushkoban for the model with both *quad* and *push* predicates. Numbers on each cell indicate attention weights, also visualized by highlighting. The red box is the goal object, the green cell is the player, and red outlines mark the goal location. Images are shown in reading order (left to right, then top to bottom). The interesting result is that in the model shown here attention is assigned in an interpretable sub-goal structure.

reveal limitations, and further in-depth studies could help to better understand these failure cases.

5.3.3 Expressivity

The results in Table 5.2 show promising findings regarding derived predicates. We firstly focus on comparing test results with and without the derived predicates *push* and *quad*.

Adding *push* has a significant effect on train coverage, increasing it from 89.5% without the *push* predicate to 95.5% with *push*. This difference is consistent even when comparing the use of attention and *push* versus attention alone, suggesting that the effect is not due to noise. This result supports our hypothesis that incorporating *push* enhances the model's ability to leverage additional logical features by providing structural knowledge of actions, without explicitly encoding the preconditions necessary for their execution.

In contrast, the *quad* predicate shows no improvement in test coverage. When combining both *quad* and *push*, test performance mirrors using *push* alone, reinforcing that *push* has a strong impact while *quad* contributes little. Generalization coverage shows a similar pattern: *push* improves performance by 8% compared to the baseline. Interestingly, this difference is almost entirely mitigated when attention is used, resulting in similar performance for models with and without *push*. A generalization performance of 82.0% presents a more or less consistent ceiling on generalization performance for all models. A possible explanation is that the limiting factors in this set do not stem from expressivity, but rather from novel, unseen challenges in the generalization set or from issues inherent to a purely value-based approach, including failure in out-of-training plan lengths.

Regarding *quad* on the generalization set, it performs poorly, achieving only 62.5% coverage. With attention, this coverage improves to 84.0%, making it the best-performing model. The poor performance of *quad* without attention is likely due to the additional atoms in the generalization set acting as noise, which is manageable during training but scales significantly on the generalization set, reducing performance.

When both *quad* and *push* are used, generalization performance is strong even without attention, suggesting attention has minimal impact in this scenario. One hypothesis is that combining *push* and *quad* allows the model to use the *push* atoms to mitigate noise introduced by *quad*. While the combination shows better generalization than either predicate individually, the exact mechanism behind this synergy remains unclear.

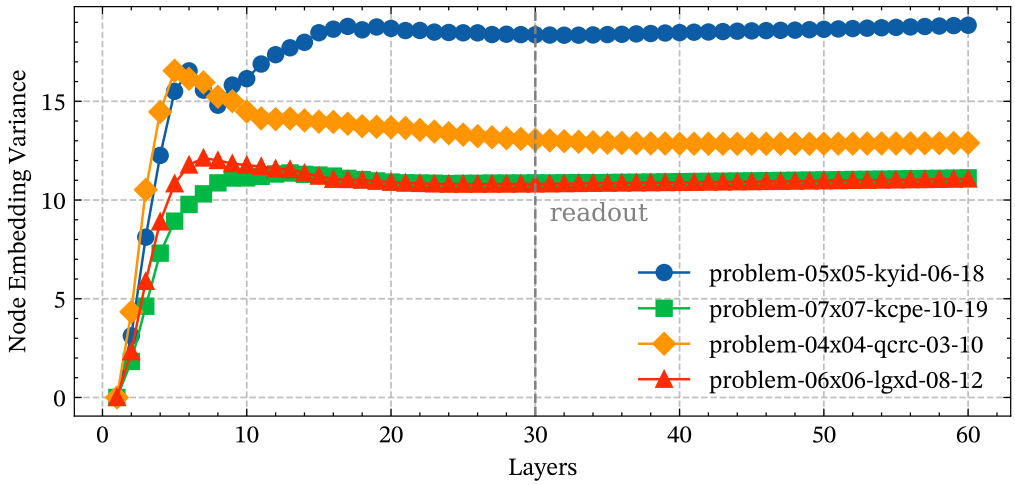
Compared to the default *adj* and adj_2 encodings, the directional encoding exhibits no relevant differences in training or test coverage but demonstrates inferior generalization. Even with attention mechanisms, performance remains comparable to the baseline without attention. Effective utilization of directional encodings may require the model to distinguish between the four objects representing the directions, which could have been achieved using complex

connections of the objects and their relative positions to the walls. An alternative explanation is that adj_2 enables the model to propagate messages over fewer hops by skipping intermediate objects. To investigate this further, we trained a new model with directional encoding and attention, using a layer size of 60. The generalization performance was in fact worse, suggesting that the factors underlying the poor results are more complex than just the layer size limitations. These findings indicate that the directional encoding is suboptimal and that incorporating additional derived predicates could improve generalization. In the subsequent *PushWorld* environment, we relied exclusively on these directional encodings, which, based on these results, appear to scale less effectively than the encoding adj_2. Due to time constraints, we did not pursue additional derived predicates for the directional encoding setting further, but it represents an interesting avenue for future research on generalization in grid-based domains.

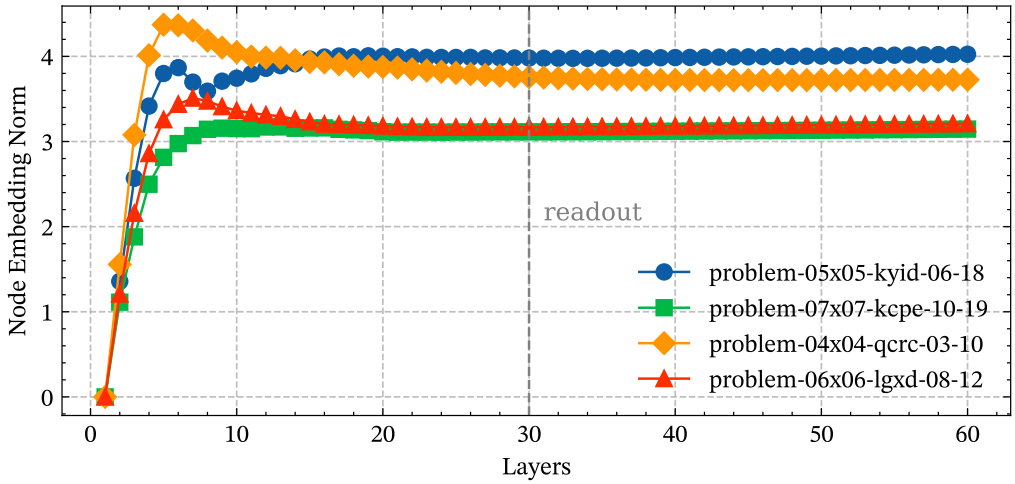
5.3.4 Progressive Training

The results in Table 5.2 are not fully representative of the progressive training method. The main takeaways are: first, using progressive training with a fixed layer size at test time does not seem to negatively affect performance, as we achieve results comparable to the baseline. Second, progressive training appears to improve generalization, albeit modestly. This is likely because the model effectively learns to regularize its embeddings, reducing both variance and norm, which could mitigate the negative impact of out-of-distribution inputs during readout. We illustrate this regularizing behaviour in Figure 5.2, which uses the same data as in Section 4.2.1 Figure 4.3, but with a model trained using a progressive loss. While the generalization of the progressively trained model is promising, we observe that the additional use of attention has minimal impact on the generalization performance. We hypothesize that the same artifact of regularization which boosts performance without attention may inhibit generalization when attention is used, since embeddings are more strongly bounded since the readout can occur at any time.

The core strength of the progressively trained model is its ability to adapt the layer size and to perform well with a higher layer count at test time. The Pushkoban domain, however, is not ideal for testing this capability: even the 15×15 grids are adequately captured with $L = 30$. Thus far, we have not encountered any limits due to the layer-size-specific bound. To evaluate the effectiveness of progressive loss training compared to fixed-layer-size training, we test performance on problem instances where computational complexity is likely constrained by the number of layers. For this, we select the plain Sokoban domain. This choice is motivated by computational efficiency, as not encoding walls as objects allows scaling to larger grids cheaper than in the Pushkoban domain. We consider problem sizes ranging from 10×10 to 19×19, using the same domain as default Pushkoban but without normal boxes, additional walls, and a single goal. We train a model using a progressive loss centered around 30 layers and also train baseline models with fixed layer sizes of 10, 30, and 60. Training is done on 1000 randomly



(a) Layer-wise variance.



(b) Layer-wise embedding norm.

Figure 5.2: Layer-wise variance and embedding norm of the R-GNN on four Pushkoban problems from the training set. This model corresponds to the one presented in the third-to-last row of Table 5.2. Compared to training without the progressive loss, both the variance and the norm remain stable across layers, staying well-regularized.

generated Sokoban problems. For evaluation of the progressively trained model we compare against each fixed-layer model, with the test-time layer count in the progressive model set to match that of the corresponding fixed size model. The results are shown in Table 5.3.

These results do not fully align with expectations, complicating the derivation of definitive conclusions. Nevertheless, several noteworthy observations emerge. First, among the fixed-layer models, a layer size of $L = 10$ appears insufficient to capture the Sokoban domain, as larger layer sizes yield improved performance. Second, results with $L = 60$ are slightly worse than those with $L = 30$, indicating that $L = 30$ is adequate to solve this problem set. This initially seems counterintuitive: in the worst case, passing a message and receiving it again

Model	Layer Size	Metrics		
		Train Coverage	Test Coverage	Test PQ
Fixed10	10	93.8	92.0 (184/200)	1.02
Progressive	10	80.0	83.5 (167/200)	1.06
Fixed30	30	96.6	97.0 (194/200)	1.01
Progressive	30	93.5	94.0 (188/200)	1.02
Fixed60	60	94.2	94.5 (189/200)	1.01
Progressive	60	91.0	93.0 (186/200)	1.02

Table 5.3: Comparison of fixed-layer and progressive models by layer count. Metrics include training and test coverage, as well as test policy quality (PQ). The same progressive model is evaluated against three fixed-layer models: Fixed10, Fixed30, and Fixed60.

across a 19×19 grid would require $(19 + 19) \cdot 2 = 76$ layers. However, accounting for the adj_2 predicate, only $(9.5 + 9.5) \cdot 2 = 38$ layers are needed. Moreover, this represents the worst case; if distances only need to propagate in one direction, only $9.5 + 9.5 = 19$ layers are required. This aligns with the observation that the model successfully solves most problems within $L = 30$.

The progressively trained model performs reasonably across different layer-size readouts. While its performance is slightly lower than that of the fixed-layer models, it remains stable. Notably, performance drops marginally at $L = 60$ compared to $L = 30$, with a decrease of 1% and three fewer problems solved at test time. Unfortunately, we did not observe improvements when scaling the layer size at test time. Further investigation could be valuable, but the main takeaway is that $L = 30$ suffices for even large problems of sizes 19×19.

5.3.5 Pushkoban Summary

To summarize the Pushkoban ablation study, several clear trends emerge. Adding derived predicates notably improved the expressiveness of the R-GNN, with *push* proving particularly useful. While *quad* alone contributed little, combining it with *push* resulted in strong performance on both the test and generalization sets. Attention readout was overall highly effective, consistently enhancing generalization without degrading performance, aside from minor computational overhead. Progressive training, while conceptually promising, showed limited benefit in this domain and did not meaningfully improve performance. Overall, AV* with R-GNNs, derived predicates, and attention provides robust learning, strong generalization, and near-optimal plan quality, whereas some alternative approaches, such as progressive loss or directional encodings, may require further refinement or additional study.

5.4 PushWorld

Now that we have seen promising results in the Pushkoban case the challenge in *PushWorld* is increased. In this case the added difficulty stems from consecutive pushing, objects of different shapes, multiple goals and combinations of walls and obstacles. This is a significant step up in difficulty and that is why we will focus primarily on the level0 puzzles [25]. In a first step we compare our AV* results with using both the R-GNN and a CNN backbone to the results presented by Kansky et al. [25]. In a second step we aim to show the difference in generalization of the CNN approach compared to the symbolic R-GNN. In the third section, we compare our approach to the classical planner LAMA. We report the number of problems solved, the number of nodes expanded, and the resulting plan quality.

5.4.1 PushWorld Results

We present the results of our approach and the central results of this thesis in table Table 5.4. This table shows the coverage on the leval0 *PushWorld* puzzles compared to the approach of Kansky et al. [25]. We present our results both with the AV* + R-GNN backbone and with a AV* + CNN backbone in order to provide a more direct methodological comparison. For the AV* + R-GNN we used the encoding as presented in Section 4.4, that is using the directional encoding and without any additional predicates. For the R-GNN we use the same setup as in Section 5.1, with the addition of using attention readout and an embedding size of 64. As for the exact methodology for the CNN we refer to Section A.2, in general we attempted to align architecturally with the CNN model used by Kansky et al. [25]. To put these numbers and results into perspective it has to be noted that we employed a model-based architecture whereas the results presented by Kansky et al. [25] employed a model-free architecture. This means that they attempt to predict action probabilities $\pi(a|s)$ and action-value functions directly $Q(s, a)$. In contrast we assume knowledge of the environmental dynamics and predict the transition via the next state directly (i.e. $\pi(s'|s)$). Additionally, unlike the AV* approach, the authors employ a reward function. They assign a positive reward for placing a goal box on a goal, provide an additional reward for reaching a goal state, and impose a negative reward for moving goal boxes away from their target positions.

In Table 5.4 we can clearly see that in all problem sets the methods using AV* clearly outperform the model-free ones on both the train and test coverage. The most striking results are seen when comparing the “multiple goals” setting. PPO achieves only 5.3% train coverage whereas the AV* + R-GNN achieves 64.5% train coverage in the same problem set. In general both train and test coverage is high for AV* + R-GNN in all settings except for the two hardest, “multiple goals” and “all”, where also all other models seem to struggle. The model-free approaches exhibit a substantial drop from training to test performance, indicating signs of overfitting. This drop curious as the train and test problems are sampled from the same distribution. In

	PPO [25]		DQN [25]		AV^* + CNN		AV^* + R-GNN	
Level 0 Set	Train (%)	Test (%)	Train (%)	Test (%)	Train (%)	Test (%)	Train (%)	Test (%)
Base	88.5	40.4	24.9	19.5	93.0	89.0	99.5	95.5
Larger Puzzle Sizes	93.2	71.5	61.2	61.8	95.0	95.0	99.0	99.5
More Walls	77.9	23.3	18.5	13.1	87.5	86.5	98.5	97.0
More Obstacles	64.7	21.4	17.6	13.1	86.0	82.0	99.5	93.5
More Shapes	74.5	24.2	23.9	17.1	90.0	87.0	98.0	90.5
Multiple Goals	5.3	1.1	0.9	0.2	36.5	37.5	64.5	54.5
All	21.2	5.7	11.4	10.1	55.0	42.5	69.5	55.5

Table 5.4: Comparison of PPO, DQN [25], AV^* + CNN, and AV^* + (R-GNN) on Level 0 sets. Train/Test (%) indicate coverage on this problem set.

contrast this train-test gap is not that big of an issue for the models trained with AV* and as such does not seem to be the result of using the CNN backbone but may indicate other issues (such as not training until convergence). In fact, the AV* + CNN approach performs competitively compared to AV* + R-GNN. While AV* + CNN does not achieve the same coverage rates, it exhibits more consistent performance from training to generalization. The largest train-to-test gap for AV* + R-GNN occurs in the "multiple goals" category, with a drop of approximately 15%. This can be attributed to the longer plan lengths required to solve these problems. Although R-GNN provides clear benefits, AV* itself still has inherent limitations. Interestingly, rerunning the data acquisition pipeline using the final value function results in solutions for 98.5% of the "multiple goals" training set. One thing that could play a role in this setting is that of expressivity, here arguing over multiple goals might be intractable within the bounds of $\mathcal{C}_2$ expressivity. A deeper investigation into the multi goal setting would be interesting. As for the "all" case, if we split the test problems into those that admit a single goal and those that admit two goals we get the following solve rates: 97/138 (70.3%) in case of a single goal and only 14/62 (22.6%) in case of two goals. This indicates that in the "all" setting, the primary difficulty does not arise directly from the combination of all problem sets. Instead, the main factor contributing to the suboptimal performance is the combination of larger grid sizes and multiple goals, which results in even longer plans.

5.4.2 Logical Generalization

For the *PushWorld* results, it is not so clear how well R-GNNs generalize. Kansky et al. [25] did not provide an explicitly harder generalization set. However, we can use cross-evaluation as another form of generalization. Given the seven models, we can evaluate their performance on the test sets of the other problem domains, providing insight into how well R-GNNs can logically extrapolate in *PushWorld*. This is particularly informative for assessing whether a model that has never encountered different shapes can generalize to novel shapes and larger problem sizes. We present these cross-validation results for all AV* + R-GNN models (Table 5.5) and all AV* + CNN models (Table 5.6). We expect that AV* combined with R-GNN is less

Trained On	Tested On						
	Base	Larger Puzzle Sizes	More Walls	More Obstacles	More Shapes	Multiple Goals	All
Base	95.5	93.0	95.0	90.0	75.5	10.0	37.5
Larger Puzzle Sizes	91.0	99.5	92.5	85.0	73.0	8.0	27.0
More Walls	96.5	94.0	96.0	90.0	45.0	6.0	14.0
More Obstacles	96.0	92.0	94.5	93.5	70.5	13.0	39.0
More Shapes	83.0	80.0	80.5	79.0	90.0	11.5	43.5
Multiple Goals	54.5	32.5	47.5	45.5	76.0	54.5	36.0
All	83.0	96.0	82.5	77.5	89.5	22.0	55.5

Table 5.5: AV* + R-GNN: Cross-evaluation of models trained on one problem set and tested on other problem sets. Percentages are colored from red (low) to green (high) for easy comparison.

prone to overfitting and, given its relational inductive biases, should generalize more effectively across domains.

Focusing first on the AV* + R-GNN results, we observe that the "base" model performs well on simpler domains, such as those with larger sizes, additional walls, and obstacles. This performance is consistent with the prior results on Pushkoban (Table 5.2), where R-GNN models demonstrated strong generalization to larger grid sizes, and the inclusion of an extra wall or obstacle did not substantially challenge the model. The real challenge arises in handling "more shapes", "multiple goals", and the "all" problem set.

The relatively strong performance on "more shapes" (75.5%) can, in part, be explained by the simplicity of the setting: the 5×5 grid restricts puzzle complexity, and many instances can be solved in only a few steps. Nevertheless, achieving 75% coverage is still notable, as the model has never encountered differently shaped objects and must infer how multi-cell objects move in order to solve the puzzle.

For "multiple goals", performance is low with only about 10% of problems solved by the "base" model, which is expected given the higher complexity of these problem instances. Even the best model in this category achieves only 54.5% coverage. For the "all" case, the "base" model solves about 37.5% of problems, which is strong given that the main model trained on this set only achieves 55.5%. This is consistent with prior strong performances on the individual subsets that make up the "all" set, combined with the fact that about half the problems include only a single goal.

Interestingly, the "multiple goals" model struggles when evaluated on single-goal problems, which at first seems counterintuitive. Conversely, it performs well on the "more shapes" scenario, likely because these problems resemble multi-goal problems, where each object effectively has multiple goal positions to reach. The "all" model performs strongly across most

Trained On	Tested On						
	Base	Larger Puzzle Sizes	More Walls	More Obstacles	More Shapes	Multiple Goals	All
Base	89.0	63.0	91.5	84.5	63.5	7.5	11.5
Larger Puzzle Sizes	84.5	95.0	85.0	77.0	59.5	2.5	11.0
More Walls	88.0	80.0	86.5	81.5	67.5	12.0	17.5
More Obstacles	88.5	68.0	87.5	82.0	59.0	4.5	12.5
More Shapes	77.5	57.5	76.0	68.0	87.0	5.5	35.0
Multiple Goals	81.0	64.5	77.0	74.0	76.5	37.5	20.5
All	60.0	65.0	56.0	55.5	80.0	19.0	42.5

Table 5.6: AV* + CNN: Cross-evaluation of models trained on one problem set and tested on other problem sets. Percentages are colored from red (low) to green (high) for easy comparison.

problem sets, as expected, but does not outperform the "base" model on "more shapes" or "larger puzzle sizes". This highlights a possible limitation in model expressivity, especially since both the "multiple goals" and "all" models did not fully converge on their training sets.

A comparison of the AV* + R-GNN models (Table 5.5) with the AV* + CNN models (Table 5.6) yields several insights. CNNs generally exhibit poor generalization to larger grid sizes. Notably, the "all" CNN model, which performed best in the R-GNN setting, shows the weakest cross-evaluation performance among the CNN models. In most cases, it is outperformed by the "multiple goals" CNN model, which was the poorest generalizer in the R-GNN case.

Despite their overall weaker performance, CNNs demonstrate some capacity for logical generalization. For instance, the model trained on "more walls" performs competitively on "multiple goals" problems, even surpassing certain R-GNN models, despite performing significantly worse on its original domain compared to its R-GNN counterpart. Furthermore, some specialized CNN models are outperformed in their own domain by the "base" model, as observed for "more walls" and "more obstacles," indicating overfitting, which is a phenomenon less pronounced in R-GNN models.

5.4.3 PushWorld Level 1

Now that we have trained a deep learning model for the Level 0 puzzles, we can use it to attempt solving the actual *PushWorld* challenge. Level 0 serves as a primitive training ground compared to the complex, hand-crafted problems of Level 1. While Level 0 is limited to grids of size 10×10, many Level 1 instances are defined on grids larger than 12×12. The shapes are also significantly more complex: whereas Level 0 only included polyominoes of three blocks, this is no longer the case for Level 1, where objects can be much larger and thus exhibit more complex shapes and interaction effects with the environment.

We refrain from training a new model directly on these problems, we assume that such models would overfit due to the limited dataset of only 68 puzzles and thus will not scale generally. Out of these 68 puzzles, 5 contain a new type of environmental interaction, the *agent wall*. These walls mark cells that the agent cannot cross, but which are passable for other objects when pushed. This dynamic is absent in the Level 0 puzzles and would require reasoning over a previously unseen predicate, something we can not expect from our R-GNN models. Therefore, we only consider the remaining 63 puzzles, which are built from the same building blocks as our comparatively simple Level 0 "all" problems.

Kansky et al. [25] report that they trained both a DQN and a PPO model on the Level 1 problems. These achieved less than 1% coverage (no puzzle was consistently solved) and 6% coverage, respectively. Running the model trained on the "all" set on these problems, we achieve a solve rate of 11.1%, corresponding to 7 out of 63 puzzles solved directly. A caveat is that the solved problems were relatively simple in terms of grid size, shapes, and goals. Nevertheless, compared to PPO, which was trained specifically on these problems, this is already a strong result.

In the next step, we evaluate how well the "all" + R-GNN model performs when used as a heuristic for A* search (*PushAV**), with a maximal nodes expansion set to 200,000. Specifically, we compare *PushAV** against the classical planner LAMA. To ensure a fair comparison, we enforce a time limit of 30 minutes per problem, for both *PushAV** and LAMA. A summary of these experimental results is provided in Table 5.7, while a full detailed breakdown of every single Level 1 problem and the performance of *PushAV** compared to LAMA can be found in Table A.1.

*PushAV** solved 84.1% of the problems (52 out of 63). Notably, it solved the problem *Irrelevant Obstacles*, for which LAMA did not find a solution within the given time limit. In one problem, *PushAV** produced a plan requiring fewer steps than LAMA, and in 26 problems the solution length was identical. From now on we are only considering results for the problems solved by both LAMA and *PushAV**. The *PushAV** total search time amounted to 1 hour and 37 minutes, while LAMA required 1 hour and 54 minutes.

Since LAMA operates on the original *PushWorld* PDDL, the number of expanded states must be interpreted with caution. More than half of LAMA's expansions correspond to states with a single available next state, where only one "push" action is possible. Consecutive push actions further inflate this count. To provide a rough lower bound, we divide the number of expanded states by three to account for these consecutive pushes. After this adjustment, LAMA expanded about 19 million states on solved problems (originally 57 million). In contrast, *PushAV**'s search expanded only 39,677 states (roughly a factor of 475 fewer).

For Level 2 and higher, the "all" model fails to solve any problems using a greedy strategy, and even with search, no problems are solved within a reasonable time limit of 30 minutes.

Model	% Solved	States Expanded	Solution Length	Wall Time	PQ
AV* + R-GNN	82.5% (52/63)	39k	1373	5856s	1.133
LAMA	93.7% (59/63)	19,000k	1286	6865s	1.0

Table 5.7: Comparison of AV* + R-GNN and LAMA on Level 1 puzzles. Metrics include percentage solved, states expanded, solution length, wall time, and plan quality compared to LAMA (PQ).

Compared to Level 1, these puzzles involve more complex shapes, larger grids, additional goals, and require substantially longer plans. In this setting, the domain knowledge acquired from Level 0 puzzles is insufficient. In particular, the value-based constraint that the model can only reliably predict plan lengths within the bounds seen during training becomes a major limiting factor. We would therefore expect policy gradient methods to perform better in this scenario.

Overall, the results of *PushAV** are highly encouraging and approach the performance of LAMA. With further refinements, such as action encodings in the R-GNN, batched A*, additional derived predicates, potential integration of a policy gradient method, and an improved training set, this approach shows strong potential for scaling. A natural long-term objective is to consistently discover shorter plans in less time than classical solvers across all *PushWorld* problems and similarly challenging domains.

6 Conclusion

In this chapter, we conclude our work on AV* and R-GNNs in *PushWorld*, reflecting on both their limitations and successes. We first summarize the key results and contributions, then discuss remaining challenges and propose directions for future research. The chapter concludes with final remarks on the significance of our findings for scalable learning-based planning and symbolic reasoning.

6.1 Summary

In this thesis, we systematically investigated the limitations of current general planning approaches and proposed novel methods to address them. Our analysis revealed that existing methods do not scale well, relying heavily on classical planners or fully expanded state spaces, and are constrained by fixed rounds of message passing. Furthermore we could show that in grid based domains robust generalization capabilities to larger grid sizes were lacking.

Contrary to prior assumptions, we demonstrated that learning general policies in *PushWorld* is indeed feasible. In particular, we established that AV* significantly boosts the acquisition of learning signals compared to purely model-free approaches. Within the simpler Pushkoban setting, we showed that the use of suitable derived predicates can enhance expressivity, although in most cases the available expressive power already suffices to solve the majority of test problems.

To improve scalable relational reasoning, we proposed an extension to the R-GNN architecture by incorporating attention-based readouts. This modification consistently improved generalization performance. In a comparative study against convolutional models, we showed that R-GNNs leverage their inherent relational structure and scale more effectively, outperforming CNN-based baselines.

Finally, we demonstrated that using a learned value function as a heuristic for A* enables performance close to that of classical planners, while significantly reducing the number of node expansions. We argue that scaling this approach further will likely yield consistently stronger performance than any general classical planner.

6.2 Future Work

While the results presented in this thesis are promising, several limitations and open challenges remain. First, value-based learners exhibit limited generalization, as they are constrained to plans of similar length as those seen during training. Policy gradient methods that operate directly on transitions may offer a more effective alternative. Moreover, our approach requires a large number of training problems (often more than 2,000), yet this was still insufficient for 100% generalization coverage. Inspired by Ståhlberg et al. [36] and their random walk approach, adversarial random walks, designed to generate increasingly challenging training problems while ensuring safety guarantees (the new found state should still be solvable), could improve efficiency in the number of required problem instances.

Another bottleneck lies in the inefficiency of A* rollouts. Batched A*, as suggested by Agostinelli et al. [2], may significantly improve GPU utilization during both training and inference. Furthermore, encoding actions directly into the R-GNN and defining policies over actions $\pi(s|a)$ rather than successor states $\pi(s|s')$ could provide both computational and representational advantages.

Competing with classical planners in *PushWorld* will require larger and more diverse training datasets, as well as the ability to confidently solve the "all" sets in both training and testing. This lack of solving the training set highlights a potential limitation in the expressive power of R-GNNs when handling multiple goals, which merits further investigation.

Our proposed progressive loss offered an interesting way to cope with fixed layer depth L, but conflicts with the attention readout limited its effectiveness. Future work could explore fixed-point graph embeddings to allow for dynamically extending the number of layers until convergence, thereby ensuring scalability to much larger grid sizes.

Finally, our comparisons with CNNs were quite basic. A stronger baseline would involve modern CNN decision-making architectures such as DRCs as demonstrated by Guez et al. [17], which exhibit planning-like behaviors. Incorporating state-of-the-art reinforcement learning implementations and hyperparameter optimization frameworks (e.g., `puffer.ai`[1]) would yield a more competitive and meaningful evaluation.

6.3 Concluding Remarks

PushWorld has proven itself as a challenging yet powerful benchmark for studying planning and reasoning. Despite significant progress, it remains unsolved; even state-of-the-art reasoning-based language models struggle with it. In this work, we identified the limiting factors of current R-GNN approaches and proposed systematic improvements. Through alternative

[1] `https://puffer.ai`

training algorithms, architectural adjustments, and heuristic search, we demonstrated that scalable learning-based planning is achievable in highly complex domains such as *PushWorld*.

We hope this work serves as both a foundation and inspiration for future research. Addressing the challenges outlined in this work will bring planning research closer to learning fully general policies that not only match but surpass human-crafted heuristics and classical planners in *PushWorld* and beyond. Achieving this goal would mark a major step toward effective and scalable generalized reasoning and planning, with broad implications for AI research and its applications.

A Appendix

A.1 Adapted Pushworld PDDL

```
(define
  (domain pushworld2)
  (:requirements :typing :universal-preconditions :strips
                 :conditional-effects :negative-preconditions
                 :existential-preconditions :disjunctive-preconditions)

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

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

  (:predicates
    (should-move ?obj - moveable-object ?dir - direction)
    (has-moved ?obj - moveable-object ?pos - position)
    (at ?obj - moveable-object ?pos - position)
    (connected ?from - position ?to - position ?dir - direction)
    (agent_wall ?pos - position)
    (is-agent ?obj - agent-object)   ;; needed to check agent-specific walls
  )


  (:action move-agent
    :parameters (?dir - direction)
    :precondition (and
      (forall (?obj - moveable-object ?d - direction)
        (not (should-move ?obj ?d))
      )
    )
    :effect (and
      (should-move agent ?dir)
      (forall (?obj - moveable-object ?pos - position)
        (not (has-moved ?obj ?pos))
```

```
40        )
41      )
42    )
43
44    (:action push-done
45        :parameters (?obj - moveable-object ?dir - direction)
46        :precondition (and
47          (should-move ?obj ?dir)
48          (forall (?pos - position)
49            (imply
50              (and (at ?obj ?pos))
51              (and
52                (has-moved ?obj ?pos)
53              )
54            )
55          )
56        )
57        :effect (not (should-move ?obj ?dir))
58    )
59
60    (:action push
61      :parameters (?obj - moveable-object ?pos - position ?dir - direction ?next-pos -
          position)
62      :precondition (and
63        (should-move ?obj ?dir)
64        (connected ?pos ?next-pos ?dir)
65        (at ?obj ?pos)
66        (not (has-moved ?obj ?pos))
67        (not (at ?obj ?next-pos))
68        (imply (is-agent ?obj) (not (agent_wall ?next-pos)))
69
70        ;; For every cell of ?obj, there must be a valid neighbor in ?dir
71        ;; that is not already occupied by a moved object
72        (forall (?other - moveable-object)
73          (imply
74            (and (at ?other ?next-pos))
75              (and
76                (not (has-moved ?other ?next-pos))
77              )
78            )
79        )
80      )
81      :effect (and
82        ;; Move all occupied positions forward
83        (not (at ?obj ?pos))
84        (at ?obj ?next-pos)
85
86        (has-moved ?obj ?next-pos)
```

```
87
88        ;; Mark blocking objects to be moved next, but exclude self
89        (forall (?other - moveable-object)
90          (when (and
91            (at ?other ?next-pos)
92            (not (should-move ?other ?dir))
93          )
94            (should-move ?other ?dir)
95          )
96        )
97      )
98    )
99  )
```

A.2 CNN Comparison

To compare the performance of the R-GNN trained with AV* against a CNN trained with AV*, we employ a CNN with an architecture similar to that used in the deep learning models from Kansky et al. [25]. The CNN consists of three convolutional layers with kernel sizes of 3×3, 3×3, and 5×5, strides of 3, 1, and 1, and channel sizes of 64, 128, and 256. The output is flattened and fed into an MLP with layer sizes [256, 256, 128, 1], using ReLU as the nonlinearity. The input is an image of size $\mathbb{R}^{2\times45\times45}$, where the first channel encodes the current state and the second channel encodes the goal state.

To simplify the workflow, we encode logical PDDL states from Pushworld / Pushkoban problems into images. Object names encode their cell positions (e.g., `cell_1_1`). Each object is drawn as a 3×3 pixel block, which allows distinguishing neighboring cells and handling differently shaped objects. The first layer encodes the current state, and the second layer encodes goal positions.

For data augmentation and to support reasoning over larger puzzles, we randomly place the generated image inside a 15×15 grid and apply random flips and rotations. This encourages the CNN to learn a value function invariant to such transformations, which is naturally handled by GNNs.

As an example, a simple 4×4 Pushkoban problem is encoded numerically as follows:

First channel (state atoms):	**Second channel (goal atoms):**
000000000222	111111111111
000000000252	111111111111
000000000222	111111111111
222000222000	111111111111
252000252000	111111111111
222000222000	111111111111
000222000000	111111222111
000242000000	111111242111
000222000000	111111222111
000000222000	111111111111
000000232000	111111111111
000000222000	111111111111

After padding into a 15×15 grid, this is visualized in Figure A.1. An example of a *PushWorld* problem with multiple goals is shown in Figure A.2.

For training, we used the AV* algorithm with some modifications: a buffer size of 80 batches, batch size of 64, and 16 workers instead of four. During training, we run A* with a search budget of 2048 but only added at most 64 states to the replay buffer per A* rollout, prioritizing those on the goal path and sampling the remainder randomly. While these settings were not extensively optimized, they provided stable results. AV* + CNN training is sensitive to hyperparameters, particularly due to data augmentation, which can cause the model to forget learned signals if not carefully considered.

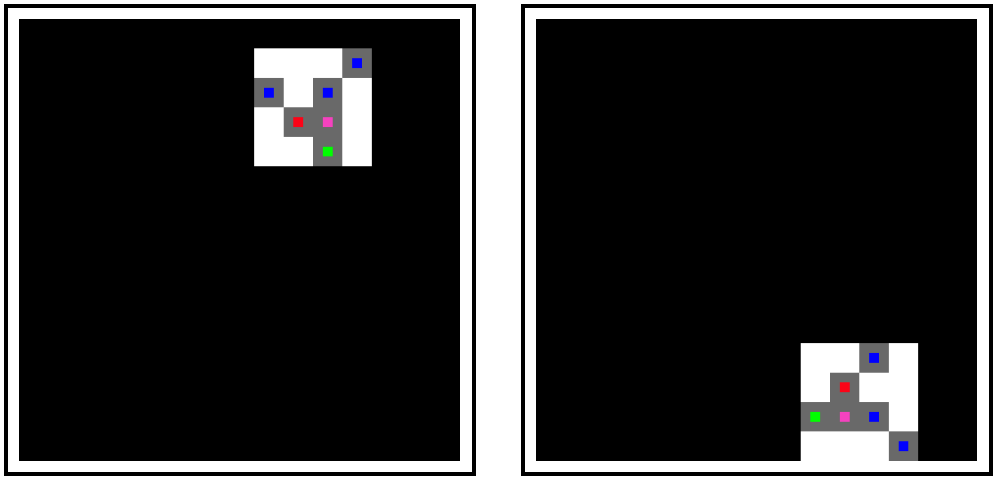


Figure A.1: Left: a CNN view of a Pushkoban problem padded to a 15×15 grid. Right: the same problem with data augmentation (random flips and rotations). Pixel encoding: 0 = white, 1 = black, 2 = grey, 3 = green, 4 = red (goal) or pink (goal position if in the second channel), 5 = blue (boxes).

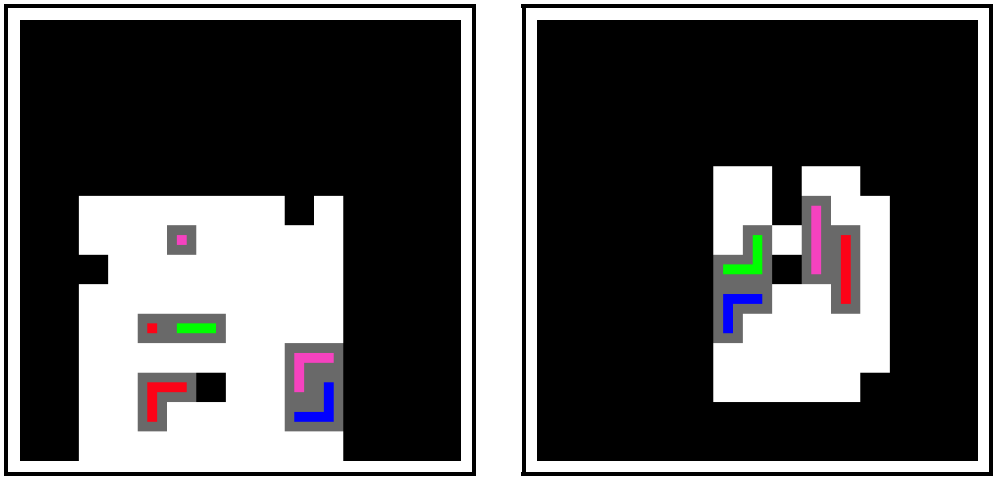


Figure A.2: CNN view of two *PushWorld* problems padded to a 15 × 15 grid. Pixel encoding as in Figure A.1.

A.3 Level 1 AV* Search Results

	*AV**					LAMA			
Puzzle	Solved	Time (s)	Plan	States	Solution	Solved	Time (s)	Plan	States
Triple Simple Tool	True	9988.37	27	71674	rrrudrurdrudrullllll lurrrrr	True	1756.28	25	31162148
Hockey Stick	False	21091.98	–	131934		True	1690.34	26	5155919
Double Trouble	True	6.27	27	31	uuuluuullllulldlldr rrrrrrr	True	1371.12	27	9800676
Building Blocks	True	4.91	19	24	druurrrruruuluuuld	True	1167.87	19	2984729
Swap Places	True	269.98	40	1569	drrrrurrrddddddurrdl lrdldlllurrluuuuurdd	True	868.12	31	2505809
Horizontal Separation	True	37.54	32	239	rrrrdruuuluurrddrur uuurullllllll	True	729.13	30	8354261
Insert Tool	True	990.23	44	6535	lddddduuuulurrrrur lddddrdrrrrulludlulu llur	True	627.03	43	4637010
Three Goals	True	20.72	42	140	urddddldrrurduuuulul ullldldldrurrdrrddrd rd	True	524.28	32	7673304
Double Simple Tool	True	1053.47	23	7905	rrrdrdruurdrudrulll urr	True	339.96	23	7374253
Ignorable Obstacles	True	2190.45	34	10930	ldlldldddlldlluuuuu uuuurrrrrrrrrr	True	314.86	34	536468
Swiss Army Knife	True	7.77	28	37	ldddllllldrrrdrdddlu uuuuuuuu	True	282.75	20	757327
A Worthy Sacrifice	True	150.38	33	1073	rrrdrrrdddrrruuuuur ullllulddrrrr	True	281.38	33	3461762

Table A.1 – continued from previous page

	AV^*					LAMA			
Puzzle	Solved	Time (s)	Plan	States	Solution	Solved	Time (s)	Plan	States
Tool Train	True	14848.25	33	88470	lullllldrrdruuuulul luuurddldrrrr	True	260.54	31	3497279
Vertical Separation	True	6.93	32	38	rdrrrruuulldrrddll llldluuuuuuu	True	153.12	30	1580528
Simple Tool	True	32.61	20	243	rrrrdrudruurudllurr	True	127.94	20	2944994
Minefield	True	8.91	31	56	lllldluuudrruullld luuuluurrdl	True	84.42	31	1414853
Multiple Goals	False	21455.49	–	84415		True	57.85	70	35637
Make Room For Me	True	533.09	42	4128	ddrrrrrrullldllllu uuuruurrdllluuurrrrd ur	True	56.92	40	285426
Pummel Through	True	4.03	15	27	urdldrrrrrdru	True	44.83	7	742314
A Perfect Fit	True	13.65	35	89	uuuuurddluulllddddd druuruurrddldrr	True	24.04	28	463223
Walk Past	True	2.86	15	19	urdldrrrrrdru	True	21.35	15	414301
Interlocking Pieces	True	25.37	30	157	urddllldrurdldruurd dddlddrrrr	True	19.61	24	87547
Mini Minefield	True	5.49	23	33	ruuullrdrddlluluuuu rul	True	17.55	19	344876
Friendly Obstacle	True	257.45	37	1470	rdrrddddrddlllllld lluuurrrrrrrrrrr	True	14.72	37	36092
Carry The Bucket	True	1897.59	43	8756	rrrrrddrruuuddllll lllllllllddduluuur ddd	True	13.67	53	7007
Corner Sequencing	True	3390.71	46	21646	rddluullldrrrrrrrr rurddddddduuuuuldlld llldrr	True	12.48	42	48335
Make A Triangle	True	10.5	31	48	drrurdldldldrddrr rrdddruuuuu	True	9.09	29	22139
Plus Two Goals	True	42.08	45	243	drrrrurrdurdrddlddd rdllrdluruuluudludll urrrr	True	7.67	36	44482
Kangaroo Pouch	True	67.57	19	441	urdldrrrruruuuurrr	True	7.18	15	116926
Youre In My Spot	True	19.6	26	144	lllllldlldddruruuuu uldddd	True	6.97	26	141577
Shape Of You	True	4.8	20	25	luuulurrllurrrrurdd	True	6.89	20	22091
Youre In My Way	True	6.57	27	41	llllddluluururddddr dllluld	True	6.64	23	175534
Two Is Company	True	223.06	35	1471	rrrrrrrdllulddduulll drrurddddlddrrr	True	6.31	35	47605
Take Diversion	True	47.22	39	330	uuuuurdldrrrrrrrlll llluuuurrrrrrrddddd	True	6.25	26	161081

Table A.1 – continued from previous page

	*AV**					LAMA			
Puzzle	Solved	Time (s)	Plan	States	Solution	Solved	Time (s)	Plan	States
Box Walls	True	315.45	42	2044	dlddddrrrdruudllllluu uurrrdrrddruruuuuurru ll	True	5.35	40	146408
Trade Places	True	403.54	42	2571	ullluuluuurrrrururr rddrdldrrdllldldlddlu uu	True	4.46	35	9127
At Crossroads	True	132.31	32	877	ddddrrrrrllllldrddd rrrruuuuuuuu	True	4.41	32	104350
Cage Door	True	5.63	27	39	uurrrdrddddluuuuuur ullllll	True	4.36	27	145186
Size Limit	False	33115.21	–	119121		True	4.02	40	4632
J Tool	True	8.94	17	55	ruuuuuuulullldr	True	3.5	14	31432
Two Goals	True	10.2	19	75	llulululddrdrrrrdd	True	3.25	18	72675
Narrow Path	True	145.5	30	1071	lldldrrrrurdddurrdd lllllllll	True	2.29	30	100526
Horizontal Channel	True	45.4	21	309	lldrrrrurrddddrdlll l	True	1.86	21	66107
Dont Get Distracted	True	1.71	8	9	uurrrrr	True	1.62	8	5945
Clear The Way	True	4.74	20	34	rdrrrululdulldrrrrr	True	1.34	14	74408
Many Small Tools	True	45.77	23	306	rrrrudrrudluldllulu lur	True	1.22	23	17011
Triple Obstacle	True	12.43	19	73	drdrruudluddlluurr	True	1.04	19	38165
Tucked Away	True	4.78	19	29	ldrrrrurddddddddrdl	True	0.87	19	33062
Sand Castle	True	752.27	41	4816	drrlddrurrlurrurddd ddurrdrdlldlldllllu r	True	0.71	27	2486
Victory Road	True	3.52	16	18	uuuuuruululuuur	True	0.62	18	24018
A Tight Squeeze	True	56.06	48	392	lddrrrrrrrddruuuuuu ddddlllllllluluuuuurr rrrrrrrr	True	0.54	48	35783
Pick A Tool	True	17.36	19	124	rrrrrudlllluullurr	True	0.42	19	6352
Double Obstacle	True	7.38	17	56	ddrrruluddlluurr	True	0.29	11	11724
Road Block	True	2.43	13	16	rdrrudllurrr	True	0.23	13	19140
Take The Long Route	True	14.57	34	103	ddddrrrrruulluulurd ddddllluuuuurr	True	0.11	34	7752
Small Wins	True	1.97	12	14	uruuluuurul	True	0.09	12	4382
Single Obstacle	True	3.38	11	24	drrudllurr	True	0.06	11	3285
Obstacle Has Obstacle	True	2.88	11	24	urrdulldrr	True	0.04	11	3229
Choose Wisely	True	5.6	22	42	ullddddddrruuddlluu rr	True	0.02	10	1341
Irrelevant Obstacles	True	472.36	27	1309	ldlldlllllurrrrrrr rrrrrrr	False	–	–	0
Swap S Places	False	2215.3	–	9035	–	False	–	–	0

Table A.1 – continued from previous page

	AV^*					LAMA			
Puzzle	Solved	Time (s)	Plan	States	Solution	Solved	Time (s)	Plan	States
Pull Up	False	2514.39	–	10529	–	False	–	–	0
Pulling	False	19475.73	–	11055	–	False	–	–	0

Table A.1: Comparison of AV^* and LAMA on selected level 1 puzzles. For AV^*, columns indicate whether solved, runtime (Time in seconds), solution length (Plan), number of states nodes (States), and solution string (Solution). For LAMA, columns indicate whether solved, runtime, found plan length, and expanded states. The solution string only includes the first character of the possible action: (u)p, (d)own, (l)eft, (r)ight.

List of Acronyms

AI	Artificial Intelligence
DQN	Deep Q-Learning (Deep Q-Network)
GNN	Graph Neural Network
MDP	Markov Decision Process
MLP	Multilayer Perceptron
PPO	Proximal Policy Optimization
R-GNN	Relational Graph Neural Network
RL	Reinforcement Learning

List of Figures

List of Tables

List of Algorithms

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] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi, "Solving the rubik's cube with deep reinforcement learning and search," *Nature Machine Intelligence*, vol. 1, no. 8, pp. 356–363, 2019.

[3] A. Bansal, A. Schwarzschild, E. Borgnia, Z. Emam, F. Huang, M. Goldblum, and T. Goldstein, "End-to-end algorithm synthesis with recurrent networks: Extrapolation without overthinking," *Advances in Neural Information Processing Systems*, vol. 35, pp. 20 232–20 242, 2022.

[4] 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.

[5] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, et al., "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[6] B. Bonet and H. Geffner, "Planning as heuristic search," *Artificial Intelligence*, vol. 129, no. 1-2, pp. 5–33, 2001.

[7] T. Bush, S. Chung, U. Anwar, A. Garriga-Alonso, and D. Krueger, "Interpreting emergent planning in model-free reinforcement learning," *arXiv preprint arXiv:2504.01871*, 2025.

[8] D. Chen and S. Thiébaux, "Graph learning for numeric planning," *Advances in Neural Information Processing Systems*, vol. 37, pp. 91 156–91 183, 2024.

[9] J. Culberson, "Sokoban is pspace-complete," 1997.

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

[11] D. Drexler, S. Ståhlberg, B. Bonet, and H. Geffner, "Symmetries and expressive requirements for learning general policies," *arXiv preprint arXiv:2409.15892*, 2024.

[12] D. Feng, C. P. Gomes, and B. Selman, "A novel automated curriculum strategy to solve hard sokoban planning instances," *Advances in Neural Information Processing Systems*, vol. 33, pp. 3141–3152, 2020.

[13] D. Feng, C. P. Gomes, and B. Selman, "Solving hard ai planning instances using curriculum-driven deep reinforcement learning," *arXiv preprint arXiv:2006.02689*, 2020.

[14] 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. 3-4, pp. 189–208, 1971.

[15] H. Geffner and B. Bonet, *A concise introduction to models and methods for automated planning*. Morgan & Claypool Publishers, 2013.

[16] C. Gehring, M. Asai, R. Chitnis, T. Silver, L. Kaelbling, S. Sohrabi, and M. Katz, "Reinforcement learning for classical planning: Viewing heuristics as dense reward generators," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 32, 2022, pp. 588–596.

[17] A. Guez, M. Mirza, K. Gregor, R. Kabra, S. Racaniere, T. Weber, D. Raposo, A. Santoro, L. Orseau, T. Eccles, G. Wayne, D. Silver, T. Lillicrap, and V. Valdes, *An investigation of model-free planning: Boxoban levels*, https://github.com/deepmind/boxoban-levels/, 2018.

[18] A. Guez, M. Mirza, K. Gregor, R. Kabra, S. Racanière, T. Weber, D. Raposo, A. Santoro, L. Orseau, T. Eccles, et al., "An investigation of model-free planning," in *International conference on machine learning*, PMLR, 2019, pp. 2464–2473.

[19] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap, "Mastering diverse domains through world models," *arXiv preprint arXiv:2301.04104*, 2023.

[20] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[21] R. A. Hearn and E. D. Demaine, "Pspace-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation," *Theoretical Computer Science*, vol. 343, no. 1-2, pp. 72–96, 2005.

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[23] J. Hoffmann and B. Nebel, "The ff planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[24] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[25] K. Kansky, S. Vaidyanath, S. Swingle, X. Lou, M. Lázaro-Gredilla, and D. George, "Pushworld: A benchmark for manipulation planning with tools and movable obstacles," *arXiv preprint arXiv:2301.10289*, 2023.

[26] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[27] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 2002.

[28] Q. Li, Z. Han, and X.-M. Wu, "Deeper insights into graph convolutional networks for semi-supervised learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[29] S. McAleer, F. Agostinelli, A. Shmakov, and P. Baldi, "Solving the rubik's cube with approximate policy iteration," in *International Conference on Learning Representations*, 2019.

[30] 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.

[31] L. Orseau, L. Lelis, T. Lattimore, and T. Weber, "Single-agent policy tree search with guarantees," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[32] S. Richter, M. Westphal, and M. Helmert, "Lama 2008 and 2011," in *International Planning Competition*, ICAPS Freiburg, Germany, 2011, pp. 117–124.

[33] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[34] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al., "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

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

[36] S. Ståhlberg, B. Bonet, and H. Geffner, "Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 32, 2022, pp. 629–637.

[37] S. Ståhlberg, B. Bonet, and H. Geffner, "Learning generalized policies without supervision using gnns," *arXiv preprint arXiv:2205.06002*, 2022.

[38] S. Ståhlberg, B. Bonet, and H. Geffner, "Learning general policies with policy gradient methods," in *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, vol. 19, 2023, pp. 647–657.

[39] S. Ståhlberg, B. Bonet, and H. Geffner, "Learning more expressive general policies for classical planning domains," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, 2025, pp. 26 697–26 706.

[40] S. Stante, "Exploring the limits of relational graph neural networks in pushworld," Bachelor's Thesis, RWTH Aachen, 2024.

[41] R. S. Sutton, A. G. Barto, et al., *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1.

[42] M. Taufeeque, P. Quirke, M. Li, C. Cundy, A. D. Tucker, A. Gleave, and A. Garriga-Alonso, "Planning in a recurrent neural network that plays sokoban," *arXiv preprint arXiv:2407.15421*, 2024.

[43] J. Topping, F. Di Giovanni, B. P. Chamberlain, X. Dong, and M. M. Bronstein, "Understanding over-squashing and bottlenecks on graphs via curvature," *arXiv preprint arXiv:2111.14522*, 2021.

[44] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[45] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3, pp. 229–256, 1992.