Optimizing Priority-Based Search for Lifelong Multi-Agent Path Finding

by

Natalie Huang

S.B. Electrical Engineering and Computer Science,
Massachusetts Institute of Technology, 2024

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2025

Authored by: Natalie Huang
Department of Electrical Engineering and Computer Science
August 18, 2025

Certified by: Cathy Wu
Professor of Civil and Environmental Engineering, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Optimizing Priority-Based Search for Lifelong Multi-Agent Path Finding

by

Natalie Huang

Submitted to the Department of Electrical Engineering and Computer Science

on August 18, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

The lifelong Multi-Agent Path Finding (MAPF) problem requires planning collision-free trajectories for agents operating continuously in dynamic environments. Traditional solvers such as Priority-Based Search (PBS) use fixed branching heuristics, which can be inefficient in high-congestion scenarios. This work explores how learning-based methods can improve PBS decision-making. We develop supervised learning (SL) policies trained from high-quality beam search trajectories and reinforcement learning (RL) policies learned directly through simulation, enabling adaptive branching strategies. Evaluations on warehouse-style and Kiva-style maps with varying agent densities show that learned policies can significantly boost throughput in congested warehouse layouts, while identifying scenarios where classical heuristics remain competitive. Our findings provide guidance on solver selection based on environment layout and congestion characteristics.

Thesis supervisor: Cathy Wu
Title: Professor of Civil and Environmental Engineering

Acknowledgments

First and foremost, thank you to Han for facilitating and supervising this project and providing constant guidance and support. Thank you to Cathy, for guiding our research and always reminding us to approach challenging problems with curiosity, persistence, and flexibility. And thank you to all members of the Wu lab who participated in the discussion and work that made this thesis possible.
Finally, thank you to my friends and family for your constant support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As warehouses increasingly rely on fleets of autonomous mobile robots to fulfill orders, the problem of coordinating their motion without collisions has become a central challenge in modern automation. Multi-Agent Path Finding (MAPF) formalizes this problem as follows: given a set of agents, each with a start and goal location on a shared graph, the goal is to compute collision-free paths for all agents that minimize a global cost metric, such as makespan or total travel time [1].

Current work in MAPF includes a variety of solvers, spanning optimal approaches, bounded-suboptimal methods, and prioritized or rule-based heuristics. Each offers different trade-offs between optimality, speed, and robustness. Performance is highly sensitive to congestion patterns and map layout, making solver effectiveness extremely scenario-dependent.

In warehouse fulfillment systems, MAPF arises naturally — robots retrieve inventory from storage zones, transport them to packing stations, and return for the next task. However, warehouse environments introduce unique constraints: densely trafficked aisles, efficient order fulfillment, and the continuous influx of new tasks. These properties significantly increase the difficulty of solving MAPF efficiently and at scale. Traditional MAPF formulations, which assume a static set of agents and goals, can fall short in these dynamic settings. We refer to this setting as Lifelong MAPF [2], where agents are continuously assigned new tasks over time.

Traditional one-shot MAPF assigns a single start and goal location to each agent, while in the lifelong formulation, tasks are continuously assigned as agents reach goal locations. Lifelong MAPF introduces temporal coupling between planning episodes, where early mistakes, such as poor agent priorities or short-sighted paths, can lead to long-term congestion and throughput degradation.

In recent years, learning-based approaches have emerged as a promising direction to overcome some of these challenges [3], [4]. Supervised learning can be used to imitate good decision heuristics, such as priority orderings or branching strategies, while reinforcement learning can optimize policies for long-term efficiency under real-time constraints. Integrating learning into MAPF solvers offers the potential to adapt to specific environments, leverage historical data, and make globally informed decisions that search alone may miss.

This thesis investigates how learning can provide guidance for traditional solvers in the lifelong MAPF problem. We first examine traditional heuristics and their weaknesses, and test different learning-based approaches to improve runtime and solution quality. Our findings provide guidance on how different solvers perform under varying map layouts, levels of congestion, and operational contexts, helping to inform practical deployment strategies in real-world multi-agent systems.

Chapter 2

Related Work

2.1 Multi-Agent Path Finding

We formalize the *Multi-Agent Path Finding* (MAPF) problem as follows.

Problem Definition

A MAPF instance is defined by:

- **Environment.** The environment is modeled as an undirected graph $G = (V, E)$, where V is the set of vertices representing discrete locations and $E \subseteq V \times V$ is the set of edges connecting neighboring locations. In this work, we consider two-dimensional, four-connected grid maps, where each traversable vertex has up to four neighbors (up, down, left, right). Obstacles are represented as non-traversable vertices.

- **Agents.** A set of k agents $A = \{a_1, a_2, \ldots, a_k\}$ is given. Each agent a_i has a start location $s_i \in V$ and a goal location $g_i \in V$, with $s_i \neq g_i$.

- **Time Model.** Time is discretized into uniform timesteps $t = 0, 1, 2, \ldots$. At each timestep, an agent may:

    1. Move to an adjacent vertex $(u, v) \in E$, or

2. Remain at its current vertex.

- **Paths.** A path for agent a_i is a sequence

$$\pi_i = \langle v_0^i, v_1^i, \ldots, v_T^i \rangle$$

where $v_0^i = s_i$, $v_T^i = g_i$, and

$$(v_t^i, v_{t+1}^i) \in E \ \cup \ \{(v, v) \mid v \in V\} \quad \forall \ t.$$

Collision Constraints

A solution is *collision-free* if no pair of distinct agents a_i, a_j $(i \neq j)$ experiences:

- **Vertex conflict:**

$$v_t^i = v_t^j$$

at the same timestep t.

- **Edge conflict:**

$$(v_t^i, v_{t+1}^i) = (v_{t+1}^j, v_t^j)$$

at the same timestep t.

Lifelong MAPF

In the lifelong MAPF setting, agents are continuously assigned new goals over an extended time horizon. Once an agent reaches its current goal, it is immediately assigned another task, often corresponding to a new pickup or delivery location. While the traditional one-shot MAPF objective is to minimize the total travel time(flowtime) or the final task completion time(makespan), lifelong MAPF objectives usually aim to optimize long-term performance metrics such as throughput while continuously maintaining collision-free paths.

2.2 Existing MAPF Solvers

2.2.1 Overview

MAPF is NP-hard in general [5]. Existing solvers take many different approaches, each with different trade-offs in terms of optimality, runtime efficiency, and scalability.

- **Rule-based approaches.** Methods such as *Prioritized Planning* [6] assign a fixed priority ordering to agents and plan agents from high to low priority. Lower priority agents must avoid collisions with the already-planned paths of higher priority agents. They are computationally efficient and memory-light but are incomplete in general, as poor priority choices can result in deadlocks or unsolvable plans.

- **Compilation-based approaches.** These methods reduce MAPF to other well-studied problems such as *Boolean Satisfiability (SAT)*, *Integer Linear Programming (ILP)*, or *Constraint Satisfaction Problems (CSP)*. Examples include MDD-SAT [7] and ILP formulations. Such methods can guarantee optimality when using exact solvers, but they typically struggle to scale beyond small- to medium-sized problem instances.

- **Search-based approaches.** These explicitly explore the joint configuration space of all agents:

  - *Conflict-Based Search (CBS)* [8]: A two-level search framework that incrementally resolves conflicts by adding constraints and replanning only affected agents. Optimal in its basic form, with many improved variants.

  - *Priority-Based Search (PBS)* [9]: Maintains a partial ordering of agents via a priority graph and resolves conflicts by adding directed priority constraints.

Rolling-Horizon Collision Resolution (RHCR)

The solvers above are used to plan solutions for a one-shot MAPF instance. In lifelong scenarios, a framework called *Rolling-Horizon Collision Resolution* is often used and is compatible with different one-shot solvers such as PBS [10]. RHCR decomposes lifelong MAPF into a sequence of Windowed MAPF instances and replans paths every h timesteps, where h is referred to as the planning window. Thus, a windowed MAPF solution only needs to be collision-free for the execution window w steps, where $w \geq h$. This approach yields similar throughput as using the entire time horizon, but with significantly smaller runtime and better scalability.

2.2.2 Priority-Based Search in Depth

Priority-Based Search (PBS) [9] is a complete and optimal MAPF solver when paired with an optimal low-level planner. In prioritized planning, agents are assigned a total priority order. The goal of PBS is to dynamically search for partial priority ordering of agents that resolves collisions. Each node in the PBS search tree is defined by a set of agent paths and a priority tree over the agents. A priority tree (PT) contains agent pairs and PBS maintains the invariant that if $a_i \prec a_j$ exists in a node's PT, there are no collisions between the paths of a_i and a_j.

The PBS search starts by generating the root node, which contains an empty priority ordering $\emptyset$. It uses the low-level planner to find an individually optimal path for every agent that respects the paths of higher priority agents. In the root node, this is the set of shortest paths for every agent since no priority pairs exist. When a conflict occurs between two agents a_i and a_j, PBS branches into two new search nodes:

1. One where a_i has higher priority than a_j ($a_i \prec a_j$).
2. One where a_j has higher priority than a_i ($a_j \prec a_i$).

For each node, PBS adds the new priority order to the node PT replans only the affected

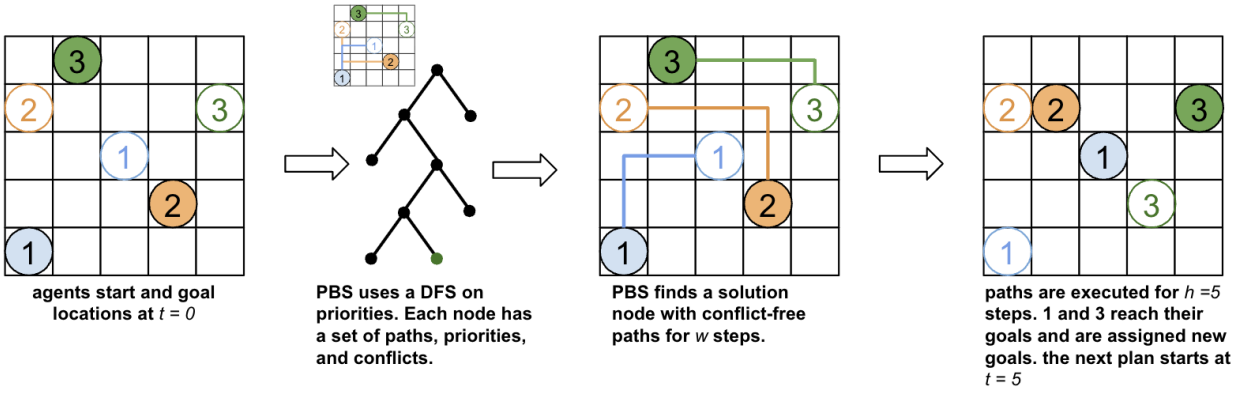

Figure 2.1: The RHCR framework visualized. Paths are planned and executed every h steps. As agents reach their goals, they are assigned new ones. The solver in this example is PBS with DFS, but another one-shot MAPF solver could be used with this framework.

agent(s) while respecting the updated priority graph. From the root node, PBS incrementally adds priority orders until a node with no conflicts is found. The high-level search among nodes used is depth-first search. PBS only backtracks when it is unable to find a solution in the current branch.

PBS is incomplete but fast compared to complete search algorithms like CBS, which motivates its popularity in larger scenarios. However, PBS lacks inherent guidance for choosing which branch to expand first. Runtime can vary significantly depending on branching heuristics. These characteristics motivate the modifications proposed in this thesis, which focus on improving PBS's search efficiency through alternative search strategies (e.g., beam search) and learning-based branching guidance.

2.2.3 Search Strategies for Frontier Expansion

High-level MAPF solvers such as PBS explore a *search tree*, where each node represents a partial solution (e.g., a priority graph and the corresponding agent paths), and edges correspond to branching decisions (e.g., imposing an ordering between two conflicting agents). The way in which nodes are selected for expansion—the *frontier expansion strategy*—has a major impact on runtime, completeness, and solution quality. Here, we introduce three

strategies.

Depth-First Search (DFS). DFS expands the most recently generated node first, diving deeply into one branch before backtracking. It uses minimal memory, as it only stores the current branch and unexplored siblings, but can spend significant time exploring unpromising regions. Standard PBS uses a DFS ordering in the priority graph space, which can lead to poor guidance in high-conflict scenarios.

Breadth-First Search (BFS). BFS expands the shallowest frontier nodes first, exploring all nodes at depth d before any at depth $d+1$. This ensures the first solution found is of minimal depth in the search tree, but memory requirements grow exponentially with depth. In MAPF, BFS can systematically explore priority orderings but is often too memory-intensive for large agent sets.

Beam Search. Beam search is a heuristic middle ground between DFS and BFS. At each depth, it evaluates all generated child nodes according to an evaluation function (e.g., $f(n) = g(n) + h(n)$), retains only the top w nodes—the *beam*—and discards the rest. This limits memory growth while focusing search effort on promising regions of the tree. Beam search is neither complete nor optimal for fixed w, but can drastically reduce runtime in large search spaces when the heuristic correlates with solution quality. In MAPF, replacing PBS's DFS with beam search can better balance exploration across different branches of the priority graph, potentially finding high-quality solutions faster.

2.2.4 Learning in MAPF

Before discussing specific learning-based methods in MAPF, we briefly outline two main paradigms used in this work as a brief introduction. *Supervised learning (SL)* trains a model to map from input features to target outputs using labeled examples, enabling it to imitate expert decisions or strong heuristics, such as branching choices in a search tree. *Reinforcement*

learning (RL), in contrast, learns a policy through repeated interaction with an environment, guided by rewards that encourage long-term performance gains.

In MAPF, RL can adapt policies to dynamic and lifelong scenarios by directly optimizing for metrics such as throughput and solution quality, while SL can leverage high-quality offline data to improve decision efficiency. Several learning-based approaches have emerged to address MAPF challenges, leveraging different methodologies to enhance planning efficiency. Huang *et al.* [4] applied supervised learning to Conflict-Based Search (CBS) by recording conflict-selection decisions from an oracle and training a ranking function to approximate those choices, leading to smaller search trees and faster runtimes. Zhang *et al.* [11] proposed learning priority orderings for prioritized planning, considering both total and partial priority models to improve success rates and solution costs. These results demonstrate that supervised policies can efficiently guide branching or ordering decisions, reducing search effort without compromising solution quality. PRIMAL [3] incorporates imitation learning and multi-agent reinforcement learning, allowing decentralized coordination based on local neighborhoods encoded by convolutional neural networks. Many learning efforts focus on increase the efficiency and scalability of MAPF solvers, but many approaches struggle to consistently outperform traditional solvers in solution quality, especially without employing extra techniques like random restarts [11].

Relation to This Thesis. In this work, we explore using supervised learning and reinforcement learning (RL) to guide branching in Priority-Based Search. Our SL approach trains a branching policy from beam search–generated datasets, while our RL approach learns search guidance policies directly from interaction, enabling adaptive improvement in dynamic and lifelong MAPF settings.

Chapter 3

Methods

This chapter presents the three main contributions of the thesis: improving Priority-Based Search (PBS) through (1) traditional search enhancements, (2) supervised learning guidance, and (3) reinforcement learning guidance.

3.1 Beam Search Priority-Based Search (BS-PBS)

3.1.1 Motivation and Overview

Priority-Based Search (PBS) traditionally performs a depth-first search (DFS) over the space of partial priority orderings. While DFS is time-efficient, it is highly sensitive to early branching decisions and suboptimal. DFS performance relies heavily on the heuristics it uses to choose node expansion order. There are two primary heuristics needed to guide the search:

1. Conflict heuristic: given a node with a variable number of collisions, how do we decide which conflict to resolve first?

2. Priority heuristic: a conflict involving agent a_1 and a_2 results in the creation of 2 child nodes, where one enforces ordering $1 \prec 2$ and the other $2 \prec 1$. How do we determine which child node is more promising?

The standard DFS implementation we benchmark against uses an earliest-conflict heuristic to choose a conflict and a lowest f-val heuristic to chose a priority ordering to resolve that conflict, where the f-val of a child node is defined as the total sum of agent path costs. These provide suboptimal but relatively good solutions in many scenarios, but in other map layouts can seriously strangle solution quality. In a lifelong scenario, one bad solution can have long-term compounded effects since each solve builds iteratively on the previous plan. One example where heuristics fail is in a warehouse, where agents must complete tasks in long and narrow aisles. As the number of agents increases, so does congestion, which often causes easy heuristics to fail to generate good solutions.

Congestion

An example of a congested scenario occurs when operating 25 agents in small 13x33 warehouse layout. We demonstrate that the f-val heuristic fails here by comparing the performance of PBS using the f-val heuristic vs choosing a child node completely randomly over 160 planning steps, or 800 timesteps. With the greedy f-val heuristic, the vanilla DFS is able to achieve slightly higher throughput for the first 200 timesteps. However, aggressive entry into aisles to complete tasks can result in congestion and deadlock and lower throughput over longer horizons, as seen in Figure 3.2. The agent layout that approximately corresponds to throughput plateau in the figure can be seen in Figure 3.1. At this timestep, a majority of agents are located densely in a few aisles, however, most are unable to reach their goals in the aisles due to the presence of agents other of agents that are either entering or exiting. Once in this state, a search with the f-val heuristic struggles to resolve congestion because the scenario necessitates that some agents need to exit the aisle, moving them away from their goal locations, in order to allow others to exit. In this scenario, PBS will find solutions where agents oscillate up and down the aisles but do not clear up the congestion- in order to do so, multiple agents would have to exit the aisle, resulting in a larger f-val solution that is difficult for the search to find.

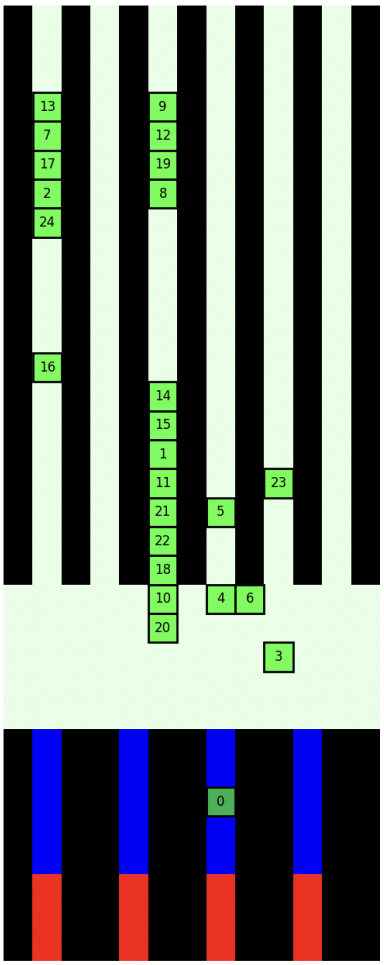

Figure 3.1: A congested 13 x 33 warehouse layout scenario. A majority of agents are trapped in aisles and are unable to complete their tasks.

These scenarios motivate us to explore alternative high-level search strategies for PBS that can balance short-term efficiency with long-term solution quality. In particular, we investigate replacing the depth-first traversal with beam search. By considering multiple candidate priority graphs in parallel, beam search reduces the risk of early, myopic decisions that lead to persistent congestion in lifelong MAPF. This broader exploration can help find priority orderings that are slightly more costly in the short term but avoid bottlenecks and deadlocks over extended horizons, thereby improving overall throughput.

Figure 3.2: Throughput of DFS with f-val heuristic vs DFS with random heuristic in a 25 agent, 13x33 warehouse. The vanilla DFS plan experiences severe congestion at step 50, causing a plateau in throughput.

3.1.2 Beam Search for Priority Graph Expansion

In BS-PBS, the high-level search operates as follows:

1. Maintain a *beam* of up to w nodes at the current search depth.

2. For each node in the beam, generate all children by resolving the next detected conflict between agents a_i and a_j. This produces two child nodes: one with $a_i \prec a_j$ and one with $a_j \prec a_i$.

3. Evaluate each child node using an f-value:

$$f(n) = g(n) + h(n),$$

   where $g(n)$ is the current solution cost (e.g., makespan) and $h(n)$ is a heuristic estimate of additional cost or remaining conflicts.

4. Sort all generated children by f-value. Ties are broken by child collision count.

5. Retain only the best w children to form the beam for the next depth level.

This process repeats until either a conflict-free solution is found or all beams are empty. Beam search is not complete due to how it discards nodes at each layer. To mitigate this, we implement an approach similar to [12], where nodes that are excluded from the beam are added to a stack and used to reinitialize the beam should it ever become empty. Unlike their implementation, the nodes in the backtracking stack are simply added in the order they are encountered, instead of sorted by a f-val, to decrease runtime overhead.

3.1.3 Complexity and Theoretical Properties

Let b denote the average branching factor of PBS (typically $b = 2$ due to binary conflict resolution), and d the maximum search depth (number of priority constraints required to resolve all conflicts).

- **Time complexity:** In the worst case, BS-PBS expands $O(w \cdot b)$ nodes per depth level, yielding a total of $O(w \cdot b \cdot d)$ node expansions. This is generally larger than DFS in shallow layers but smaller than BFS's $O(b^d)$.
- **Memory complexity:** Beam search requires storing all w nodes at each depth, for $O(wd)$ space, compared to $O(d)$ for DFS and $O(b^d)$ for BFS.
- **Completeness:** For fixed w, BS-PBS is normally incomplete but can be modified to use backtracking.
- **Optimality:** Like PBS with DFS, BS-PBS be is sub-optimal.

3.1.4 Integration into PBS

The integration of beam search into PBS requires modifying the high-level search loop:

1. Replace the DFS stack used in standard PBS with a priority-sorted list representing the beam.
2. At each depth, expand all nodes in the beam and generate their children.

3. Rank all children using the f-value and tie-breaking rules.

4. Retain only the top w children as the next beam.

The other components of PBS remain unchanged: the low-level planner still computes agent paths respecting the current priority graph, and conflicts are resolved one at a time by branching into two children, each with a different added priority constraint.

3.2 Supervised Learning for PBS Branching

3.2.1 Overview

Beam search offers promising alternatives to DFS, but suffers from long runtimes compared to DFS. Beam search runtimes roughly scale up with the width of the beam which can be hugely costly in larger maps with more agents. Our experiment attempts to learn to guide DFS using higher-quality beam search solutions as an expert reference.

3.2.2 Feature Augmentation

A vanilla PBS node contains information about full agent paths, conflicts in those paths, and the current priority tree. For each node, we tried generating extra features to help the model discriminate between children options. We reference [11], which used learning to determine a prioritized planning order, to design multiple features related to an agent's multi-valued decision diagram (MDD). MDD_i for agent i is a directed multi-level acyclic graph consisting of all shortest paths from the agent's start location s_i to goal location g_i. Level t of MDD_i contains all possible locations a_i could occupy along one of its shortest paths. We tested including the following MDD features for each agent:

1. Average width of levels: we hoped that this metric would capture how constrained each agent's path was

2. Number of unit width levels: the number of levels in an agent's MDD that only contain one possible level. An agent with many unit width levels is very constrained.

3. Start locations in MDD: the number of agents who have a start location in the current agent's mdd

4. Goal locations in MDD: the number of agents who have an end location in the current agent's mdd

However, preliminary results indicated that this did not help much in training, and the additional computational overhead of generating these metrics for every agent at every branching step was very high, so we ended up excluding these.

3.2.3 Dataset Generation

First, we create a dataset of beam-search solutions for 40 agents in a medium warehouse. Rather than maintaining all nodes expanded in the search, we only save nodes in path from the final solution node found to the root. At each level, we record the parent node, children nodes, and a label corresponding to which child node was selected in the solution path. For each node, we save full information on paths, priorities, conflicts, and f-val. If the model can accurately learn to predict which child should be chosen, this model could act as DFS heuristic that can produce beam-search quality solutions with a faster runtime.

3.2.4 Model Architecture

Node Encoder

The main part of the model is the node encoder, which produces a fixed-dimensional representation of a search node by combining scalar node metrics and structured encodings of conflicts and priorities. We outline the high-level design decisions here.

ID Invariance. In our implementation of PBS, agents are described by a unique integer ID. However, we want the trained model to be invariant to id- if two MAPF instances have the same exact start and goal locations but permuted agent ids, the representation should be the same. To do this, we discard agent ids and use the agent start location, next goal locations (up to 2), current path length, and shortest possible path length to create an agent representation. The locations are passed through a location embedding layer and the path lengths through a small MLP, then concatenated.

Conflict and Priority Embeddings. In a PBS node, each conflict is represented by two agent ids, up to two locations (for vertex and edge conflicts) and a time. Nodes can contain a variable number of conflicts. To represent the full set of conflicts for each node, we use the agent encoder described above to embed both agent ids. Then, we use the location embedding layer, which is shared with the agent encoder, to encode the location(s) of the conflict. If there is only one, a padding value is used. After each conflict is embedded, all embeddings are averaged to create the final conflict representation.

A similar approach is taken to represent all priorities. Each priority in the node's PT is represented by two agent ids. For every priority relationship, the agent ids are passed through the agent encoder layer and concatenated. We then take the average of all individual priority embeddings to produce the final priority representation.

Full Embedding. We embed scalar features relevant to the node, such as f-val and the number of collisions using a small MLP. The final node representation is obtained by concatenating the transformed scalar features, the aggregated conflict embedding, and the aggregated priority embedding. This concatenated vector is then passed through a final transformation network to produce the encoded node feature vector used by the downstream decision model.

Prediction Head

The prediction head is responsible for selecting the child node, given the parent and two child choices. The network takes the encoded parent and the difference of the first child and second child in embedding space. We use the embedding difference to better help the model directly compare two options, as well as make it invariant to child ordering. The model outputs a score representing the relative preference for the first child over the second. A positive score indicates that the first child is more likely to lead to a successful or efficient solution, whereas a negative score indicates the opposite. This score is passed through a sigmoid function to produce the probability of selecting the first child.

3.2.5 Model Learning

The model uses BCEWithLogitsLoss, which learns to minimize the loss function

$$\ell(y, s) = -\left[\, y \, \log \sigma(s) + (1 - y) \log \left(1 - \sigma(s)\right) \right]$$

over the entire dataset, where y is derived from the ground truth child label and s is the predicted model score.

3.2.6 Inference

The trained model can then be used as a heuristic to guide DFS search. At each branching decision, instead of selecting the child with lower f-val, we query the model with the current parent and children states to guide the search instead.

3.3 Reinforcement Learning for PBS Guidance

The quality of beam-search guided PBS (and therefore the supervised model that learns from its solutions) still heavily relies on heuristic performance to discover good solutions. While

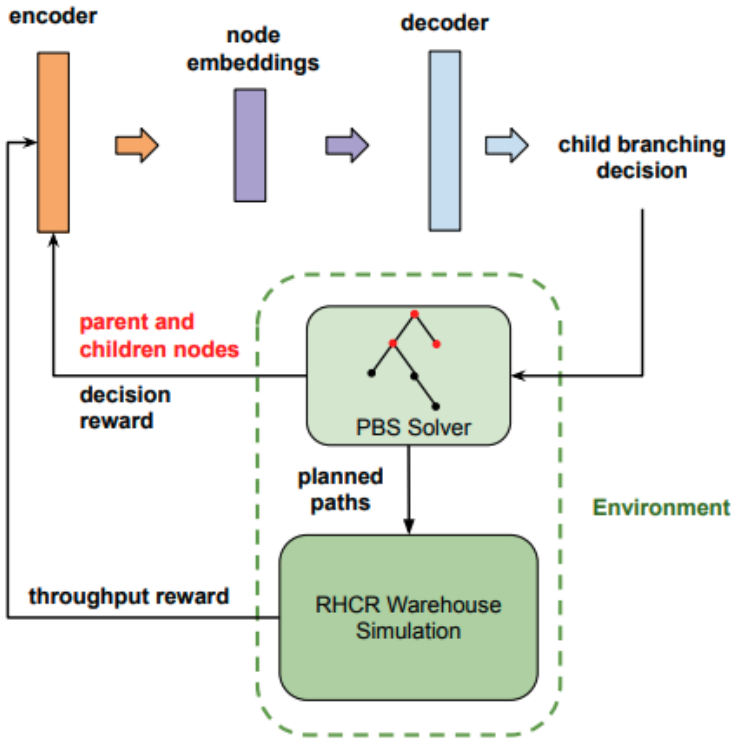

Figure 3.3: The framework of our proposed RL setup. At every step, the PBS solver inferences the model to decide which branch to take. Once a solution node with conflict-free paths is found, the warehouse simulation executes the paths. Decision-level rewards are received for each branching decision, and throughput rewards received for every execution of paths. Both are used to update the policy.

the beam somewhat mitigates the tendency of DFS to make short-sighted bad decisions, solutions are still sensitive to the f-val heuristic and the width of the beam. Additionally, neither approach captures the long-term considerations and solution dependencies that exist in lifelong MAPF. Reinforcement learning (RL) is well-suited to address the sequential search problem above, which involves not only generating paths that are good for the current agent start/goal states, but paths that will result in high-throughput lifelong plans overall. In this section, we introduce a RL-guided approach for learning a DFS branching heuristic specifically tailored for lifelong MAPF.

As an overview, we first formulate the PBS child node select problem as a Hierarchical Markov Decision Process (MDP). The RL policy is responsible for selecting the child branch at each step in the PBS search. To do this, the RL policy interacts with a wrapper environment

around our base PBS solver. The RL policy explores the current search tree until a solution node is reached. The simulation environment executes the solution paths found and returns feedback on the overall throughput, which are used to update the RL policy to iteratively explore and improve the quality of the branching heuristic. The overall framework is illustrated in Figure 3.3.

3.3.1 RL Formulation for PBS in Lifelong MAPF

We formulate the PBS decision-making problem as a hierarchical Markov Decision Process (MDP) operating at two temporal scales: individual PBS solves and the broader warehouse operation.

Hierarchical State Space

Global State (Warehouse Level): The global state S^t_{global} captures the warehouse operational context at time t:

$$S^t_{\text{global}} = (A^t, T^t, E^t, H^t) \tag{3.1}$$

where:

- $A^t \in \mathbb{R}$: Current positions of n agents in the warehouse. We use row-major format instead of a 2D coordinate system.
- T^t: Set of active task assignments and goal locations
- E^t: Environment state including congestion levels and completed tasks
- H^t: Historical performance metrics (recent throughput, solve times)

Local State (PBS Node Level): The local state S^t_{local} represents the current PBS

search context:

$$S^t_{\text{local}} = (N^t, C^t, P^t) \tag{3.2}$$

where:

- N^t: Current PBS search node containing agent paths
- C^t: Set of unresolved conflicts $\{(a_i, a_j, \ell, \tau)\}$ between agent pairs
- P^t: Current priority constraints imposed by the search tree
- S^t_{global}: Global warehouse context

Observation Space

Due to the complexity of the full state space, we design a compact observation O^t:

$$O^t = \phi(S^t_{\text{local}}) = (\phi_{\text{parent}}(N^t), \phi_{\text{child}}(N^t_1), \phi_{\text{child}}(N^t_2)) \tag{3.3}$$

Each node embedding $\phi_{\text{node}}(\cdot)$ encodes:

- **Scalar features:** Search metrics (f-value, collision count, search depth)
- **Conflict embeddings:** Neural encoding of agent-location-time conflicts
- **Priority embeddings:** Representation of current agent priority constraints

We do not explicitly use the global state for efficiency, but the agent positions and task assignments are implicit in the node path state.

Action Space

At each PBS decision point, the agent chooses between two child nodes:

$$A = \{0, 1\} \tag{3.4}$$

where:

- $a = 0$: Select the first child node (prioritizing one agent ordering)
- $a = 1$: Select the second child node (prioritizing the reverse ordering of child 1)

Transition Dynamics

The transition function operates at two levels:

Intra-solve transitions: When conflicts remain, the system transitions within the PBS search tree:

$$P(S^{t+1}|S^t, a^t) = \text{PBS}_{\text{expand}}(S^t, a^t) \tag{3.5}$$

Inter-solve transitions: When a conflict-free solution is reached, paths are executed for w timesteps, where w is the planning window in the RHCR framework. After execution, a new PBS root node is generated from the updated agent locations.

$$P(S^{t+1}|S^t, \text{solution}) = \text{Execute}_{\text{paths}}(S^t) \circ \text{Initialize}_{\text{PBS}}(\cdot) \tag{3.6}$$

Transitions are handled within the simulation environment by the low-level planner, which uses Safe Interval Path Planning [13] to find optimal shortest paths while respecting any added priority constraints, or to generate a new PBS root node after path execution.

Multi-Scale Reward Function

The reward function combines immediate PBS efficiency with long-term warehouse performance considerations:

Decision-level reward:

$$R^t_{\text{decision}} = w_1 \cdot \Delta^t_{\text{conflicts}} + w_2 \cdot \Delta^t_{f\text{-value}} \tag{3.7}$$

where $\Delta^t_{\text{conflicts}}$ represents conflict reduction between the chosen child and the parent node, and $\Delta^t_{f\text{-value}}$ measures the difference in path length between the chosen and unchosen children nodes.

Solve-level reward:

$$R^k_{\text{solve}} = \alpha \cdot \text{progress}^k + \beta \cdot \text{tasks\_completed}^k \tag{3.8}$$

where k indexes the solve number within an episode. Progress is defined as the average distance made by all agents towards their respective goal locations in the execution window w, and tasks_completed is the number of new tasks that were completed in the window w. Depending on task assignment and map size, task completion can be sparse. To check that agents are actually making progress towards goals, we look at distance traveled. However, average progress made is maximized at w, since agents can only move one grid cell every timestep, so we add the two metrics to create a stronger signal that directly correlates to throughput.

Total episodic reward: For each decision t in solve k, the total reward combines immediate and future solve-level components:

$$R^{t,k}_{\text{total}} = \lambda \cdot R^{t,k}_{\text{decision}} + (1-\lambda) \cdot \sum_{j=0}^{H-1} \gamma^j \cdot R^{k+j}_{\text{solve}} \tag{3.9}$$

where H is the future reward horizon (e.g., $H = 4$ for next 4 solves).

Parameters $\lambda \in [0, 1]$ controls the balance between immediate search efficiency and long-term warehouse throughput, while γ provides temporal discounting across the extended planning horizon.

This formulation ensures that:

- Each decision in solve k receives credit for the next H solve-level rewards
- Future solve rewards are discounted by γ^j where j is the number of solves ahead

- Decisions contributing to earlier solves receive higher total rewards due to longer future horizons
- Parameter $\lambda \in [0, 1]$ balances immediate search efficiency with multi-solve warehouse performance

We note that in standard RL, returns are calculated as the discounted sum of future rewards:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \tag{3.10}$$

However, since rewards from future PBS solves are baked into our episodic reward formulation, we treat returns as the episodic reward. Previously, we attempted to only assign the solve-level reward to the PBS solution node and use the return calculation, but found that rewards were too sparse among individual decisions.

Episode Structure

A complete episode consists of multiple PBS solves within a fixed planning window:

$$\text{Episode} = \{\text{Solve}_1, \text{Solve}_2, \ldots, \text{Solve}_K\} \tag{3.11}$$

Each solve Solve_k contains a variable-length sequence of PBS decisions:

$$\text{Solve}_k = \{(s_1^k, a_1^k, r_1^k), (s_2^k, a_2^k, r_2^k), \ldots, (s_{T_k}^k, a_{T_k}^k, r_{T_k}^k)\} \tag{3.12}$$

Episodes terminate when:

- The planning window expires (normal termination)
- A solve failure occurs. If PBS is not able to reach a solution within a generous time window, the episode is terminated and assigned a large negative penalty.

- Performance degrades below threshold. If too much congestion is detected, or if throughput is stagnant for many consecutive episodes, we terminate and assign a large negative penalty.

To expand on the third case, congestion is detected per agent by looking at their executed path in the last $2w$ timesteps and comparing it to the current path. If over a certain percentage of locations are repeated, we mark the agent as congested. This is mainly to detect the oscillatory behavior discussed in 3.1.1. If too many agents are congested, we terminate the episode.

Policy Objective

The optimal policy maximizes expected cumulative reward across the hierarchical decision process:

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R\big(S^t, \pi(O^t)\big)\right] \tag{3.13}$$

This formulation enables the policy to learn correlations between immediate PBS search decisions and long-term warehouse throughput, bridging the gap between algorithmic choices and system-level performance.

3.3.2 Policy and Value Network Architecture

The RL agent consists of separate actor and critic networks sharing a common node encoding scheme described in Section 3.2.4. At each decision step t, the environment provides the parent search node s_t and its two candidate child nodes $s_t^{(1)}$ and $s_t^{(2)}$.

Actor. The actor network receives the encoded representations of the parent node and both child nodes, $(\phi(s_t), \phi(s_t^{(1)}), \phi(s_t^{(2)}))$, where $\phi(\cdot)$ denotes the nodeLocationEncoder. These embeddings are passed through a MLP scoring head to produce a single score. Similar to the supervised learning approach, scoring head takes in the parent embedding and difference

in embedding space between the first and second child. The output s of the scoring head is converted to logits over the action space by taking $[s, -s]$. While this may limit expression, this ensures that input $(\phi(s_t), \phi(s_t^{(1)}), \phi(s_t^{(2)}))$ and $(\phi(s_t), \phi(s_t^{(2)}), \phi(s_t^{(1)}))$ will produce logits that are exact negations of each other, enforcing invariance to the ordering of the two child nodes and yielding a symmetric policy over the action space. The logits are softmaxed to create a probability distribution over the action space.

Critic. The critic network receives only the encoded parent node $\phi(s_t)$ and outputs a scalar value $V_\psi(s_t)$ estimating the expected return from the current state. Restricting the critic to the parent node ensures that the value estimate remains action-independent, providing a stable baseline for advantage estimation in PPO.

This asymmetric actor/critic observation space allows the actor to leverage comparative information between candidate actions, while the critic maintains a consistent, state-based value function.

3.3.3 Training Procedure

To train the RL policy, we use Proximal Policy Optimization, a policy gradient method that is widely used in RL for its stability and superior performance. At every iteration, rollout action and reward data are collected and used to update the policy. PPO's main innovation is a clipping of the policy update to ensure stability.

Every epoch, we collect a rollout dataset of 8 consecutive PBS solves over 4 randomly seeded environments. PPO updates are run over the entire batched dataset. Entropy loss is added to encourage exploration, especially since the action space is very limited. For stable training, we incorporate techniques such as advantage normalization and gradient clipping[14].

Validation

Validation is performed every 20 epochs on a set of 5 random seeds that are not used in training environments.

Simulation Integration

Our existing simulation environment included a PBS solver in C++, and a Gym [15] environment wrapper to handle parallelization among CPUs. However, our previous work had interacted with the simulation on an entire solve level. For every RL action taken, one entire MAPF solution is generated. This scenario differs in that every individual branching decision in the PBS solve corresponds to one agent action. To bypass the engineering effort of a full environment reimplementation, we hack the environment by adding a callback function in the PBS solver that inferences the RL model. This approach, while quick, had some downsides- our simulation was no longer parallelized, and increased runtime significantly. Ultimately, we had to collect less training data overall for the RL model.

3.4 Experimental Setup

3.4.1 Benchmarks and Maps

Experiments are run in an open-source lifelong warehouse simulation environment that implements the RHCR framework. The high-level solver is PBS and the low-level solver used is SIPP. The default settings are used, with a planning horizon $w = 5$, execution horizon $h = 5$, and a simulation window of 800 timesteps. We compare each approach against PBS running DFS with a f-val heuristic (the vanilla version), and DFS with a random branching heuristic.

We primarily test experiments in two scenarios:

1. **Amazon fulfillment center (Kiva map).** Bots start at the robot homes at the edges

and complete tasks in the center of the map at the blue squares. We use a medium-sized Kiva map, which has size 46 x 33 with low obstacle density, shown in Figure 3.4

2. **Small and medium warehouse maps.** Bots start at the bottom of the map and must alternate between aisle and inbound/outbound station tasks. The small warehouse map is 13 x 33 with high obstacle density, and the medium is 39 x 33 and is shown in Figure 3.5

Task Assignment

Task assignment is handled differently in the two scenarios. In the Kiva map, agents are initialized in home locations with a random endpoint location as their initial task. As the simulation executes planned paths, if any agent is less than w steps away from their current goal, where w is the planning window, they will be assigned a next random goal. In the warehouse map, the map is divided into inbound, outbound, deck, and aisle locations. Agents ship or receive packages from inbound and outbound aisles, and store or retrieve these packages from aisle locations. On initialization, agents are randomly placed in the blue home locations and loaded with a package. When an agent's most recent goal is from inbound, its subsequent destination is an aisle station, and the agent transitions to an unloaded state. In contrast, if the last goal is from an outbound location the next destination is chosen randomly between an inbound station and an aisle station, resulting in the agent becoming loaded. For last goal is from an aisle location, the decision depends on the agent's current loading state: if the agent is unloaded, it randomly selects between an inbound or aisle station and becomes loaded; if it is already loaded, the next destination is an outbound station, returning the agent to an unloaded state.

Solution Repairing

In larger congested scenarios, solvers are sometimes unable to find full set of conflict-free solution paths due to infeasibility or runtime limits. To mitigate this, we adopt a repair

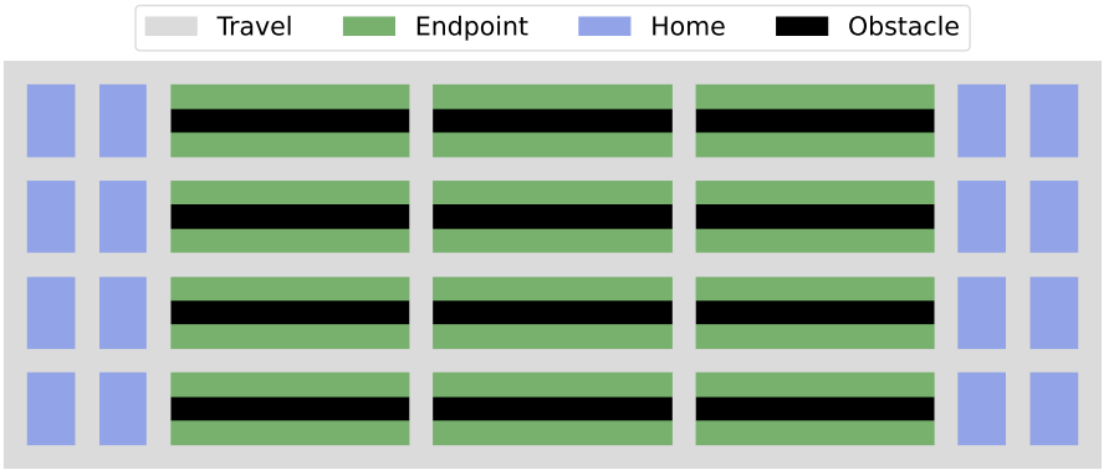

Figure 3.4: Amazon (Kiva) scenario map, with obstacle density around 15%.

mechanism from RHCR [10], which systematically resolves conflicts in the initial infeasible paths. Each conflict is resolved by adding wait actions. In the worst case, every agent stops and waits, but does not collide. Since we currently model robots as point agents with no mass or inertia, these wait actions are permitted. However, we aim to minimize repairing overall, since this may not be possible in a real-world environment. During SL dataset collection, we explicitly discard any repaired solutions. In RL training rollouts, if the solver fails to produce a solution we reset the simulation back to 0 and assign a large negative reward. During evaluation, we allow the repair mechanism to be used.

3.4.2 Evaluation Metrics

In the lifelong scenario, our main metric is **total throughput**, which is the total number of tasks completed in the entire simulation horizon of $T = 800$. Higher throughput implies more efficient task completion of continuous goals.

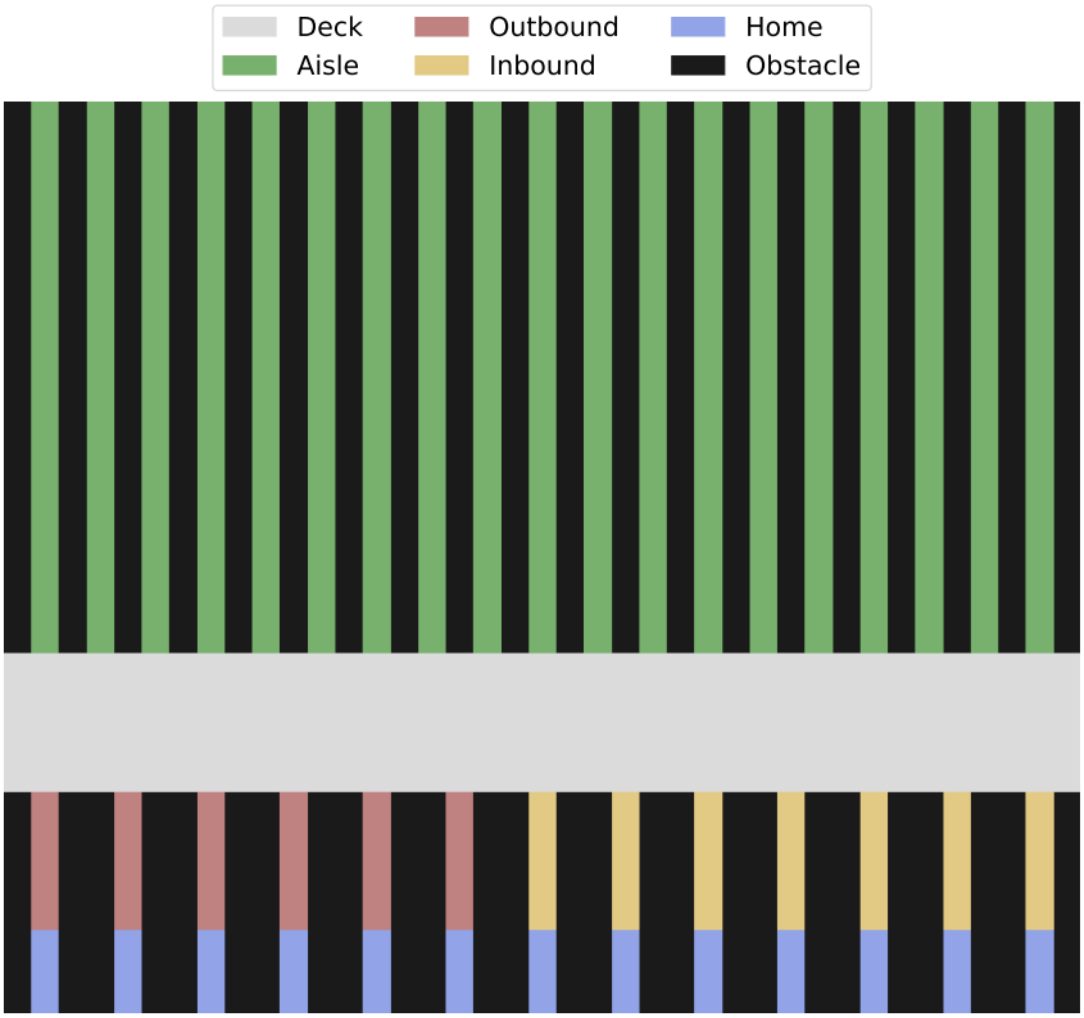

Figure 3.5: A medium warehouse scenario map, with obstacle density around 47%.

Chapter 4

Experiments

This chapter evaluates the proposed PBS variants, comparing them against classical PBS and other baselines. Our evaluation focuses on answering the following questions:

1. How does each solver perform in terms of throughput and solve time across different map layouts and congestion levels?

2. How well does each approach generalize to scenarios with varying agent densities?

The experimental setup, including environment configurations, solver parameters, and evaluation metrics, is described in Section 3.4. Here, we present results for three map types—small warehouse, medium warehouse, and Kiva—and discuss their implications.

4.1 Beam Search PBS Experiments

We run beam search in all 3 scenarios, with a variety of agent densities. Each experiment is run with 40 random seeds.

4.1.1 Runtime and Success Rate vs Beam Width

Small Warehouse Maps

Beam search yielded varying results in a small map. It only outperformed DFS with a random branching heuristic in the 20 agent scenario, but outperformed DFS with a f-val heuristic in all scenarios.

# Agents	Method	Avg Time (s)	Avg Task Acc	Avg HL Expanded	Avg Depth	Avg Repairs	Avg Cost
20	Beam (w=10, Backtrack)	13.20	**312.45**	87.40	11.66	0.00	535.40
	Normal DFS	3.52	255.70	18.15	12.78	0.00	547.66
	DFS (Random Branching)	3.28	309.80	15.40	11.76	0.00	558.48
25	Beam (w=10, Backtrack)	40.80	307.35	270.50	17.16	0.10	679.84
	Normal DFS	7.90	264.95	35.60	19.25	0.00	679.79
	DFS (Random Branching)	5.91	**322.15**	30.37	17.27	0.00	724.77
30	Beam (w=10, Backtrack)	95.33	288.05	603.15	23.10	0.17	840.54
	Normal DFS	14.45	281.93	77.55	24.60	0.07	822.31
	DFS (Random Branching)	15.17	**321.20**	86.40	22.89	0.12	894.27

Table 4.1: Performance comparison on `warehouse_small.map` with varying agent counts (20, 25, 30). Best Avg Task Accuracy for each agent count is highlighted in bold.

Medium Warehouse Maps

Compared to the small map, beam search provided consistently better results on medium warehouse maps. In the 40 agent scenario, beam search increased throughput by 7.5% and 12.5% compared to normal and random DFS, respectively. In the 50 agent scenario, beam search yielded a 10.9% increase 14.2% increase, respectively.

Kiva Maps

Beam search resulted in overall small throughput improvements across the board in Kiva scenarios, but increased runtime by 7-8x in each scenario.

# Agents	Method	Avg Time (s)	Avg Task Acc	Avg HL Expanded	Avg Depth	Avg Repairs	Avg Cost
40	Beam (w=10, Backtrack)	21.89	**780.80**	150.70	12.59	0.15	1067.04
	Normal DFS	5.44	725.70	18.07	13.22	0.00	1135.97
	DFS (Random Branching)	6.57	693.15	25.53	14.61	0.05	1136.02
50	Beam (w=10, Backtrack)	89.49	**874.10**	623.61	19.81	0.45	1404.75
	Normal DFS	10.43	788.08	48.44	21.35	0.03	1445.83
	DFS (Random Branching)	21.19	765.45	126.04	22.40	0.15	1494.54
60	Beam (w=10, Backtrack)	213.68	**996.95**	1420.41	27.19	1.35	1690.35
	Normal DFS	23.94	851.38	111.21	30.53	0.15	1751.64
	DFS (Random Branching)	55.20	822.12	322.80	31.30	0.72	1837.28
70	Beam (w=10, Backtrack)	527.98	**1050.35**	3082.13	37.29	3.20	2006.52
	Normal DFS	36.91	876.30	160.76	40.91	0.30	2142.53
	DFS (Random Branching)	127.83	902.00	722.28	40.01	2.40	2157.48

Table 4.2: Performance comparison on `warehouse_medium.map` with varying agent counts (40, 50, 60, 70 agents). Best Avg Task Accuracy for each agent count is highlighted in bold.

4.1.2 Summary

The beam search results demonstrate that especially in the medium warehouse scenario, there is definite room for heuristic improvement. We theorize that beam search worked the best in this layout because the f-val provides a good base heuristic since it outperforms random branching, and the added exploration due to beam search helps mitigate congestion. In the small map, we see that DFS with the f-val heuristic is worse than DFS with a random heuristic, suggesting that beam search, which chooses the beam based on f-val, would not improve throughput. Finally, the Kiva map shows limited gains by beam search, which we believe is due to the lack of congestion, since the map layout contains fewer narrow aisles and choke points.

4.2 Supervised Learning PBS Experiments

We chose to train a supervised learning model on beam search solutions for the medium warehouse, 40 agent scenario. We chose this specific scenario because beam search showed

# Agents	Method	Avg Time (s)	Avg Task Acc	Avg HL Expanded	Avg Depth	Avg Repairs	Avg Cost
60	Beam (w=10, Backtrack)	74.45	**1765.45**	338.49	35.88	0.00	1195.04
	Normal DFS	11.51	1759.20	35.36	35.34	0.00	1193.25
	DFS (Random Branching)	14.84	1431.60	44.02	43.43	0.00	1211.05
70	Beam (w=10, Backtrack)	131.13	**2000.38**	476.40	49.68	0.00	1395.93
	Normal DFS	18.23	1983.90	48.74	48.64	0.00	1397.40
	DFS (Random Branching)	24.71	1516.80	60.91	59.33	0.00	1432.38
80	Beam (w=10, Backtrack)	213.51	**2206.40**	660.57	65.88	0.10	1599.83
	Normal DFS	26.09	2197.30	64.54	64.31	0.00	1601.08
	DFS (Random Branching)	52.55	1570.10	112.97	78.57	0.30	1647.43
100	Beam (w=10, Backtrack)	488.16	**2540.20**	1126.36	104.33	0.60	2014.93
	Normal DFS	59.55	2515.35	116.23	103.83	0.20	2023.84
	DFS (Random Branching)	259.48	1598.50	448.27	119.02	4.55	2045.81

Table 4.3: Performance comparison on `kiva.map` with varying agent counts (60, 70, 80, 100). Best Avg Task Accuracy for each agent count is highlighted in bold.

a strong increase in throughput while keeping a manageable runtime for dataset collection compared to the small map and kiva scenarios. Beam search with 40 agents is also able to run without many repairs, while above that the number of repairs necessary begins to increase.

We collected a dataset of 10000 PBS solutions (each solution contains a full sequence of branching decisions) found using beam search. Beam search is on average better than normal DFS but not always, so during dataset collection we only included beam search solutions that outperform vanilla DFS with a f-val heuristic. Due to memory constraints, only half this dataset was used in training, which resulted in 40596 samples total. 8000 samples from the full dataset were reserved for validation.

The model was trained for 40 epochs with a batch size of 64 and learning rate of 0.001. The Node Encoder model parameters used a location embedding dimension of 32 and node embedding dimension of 64.

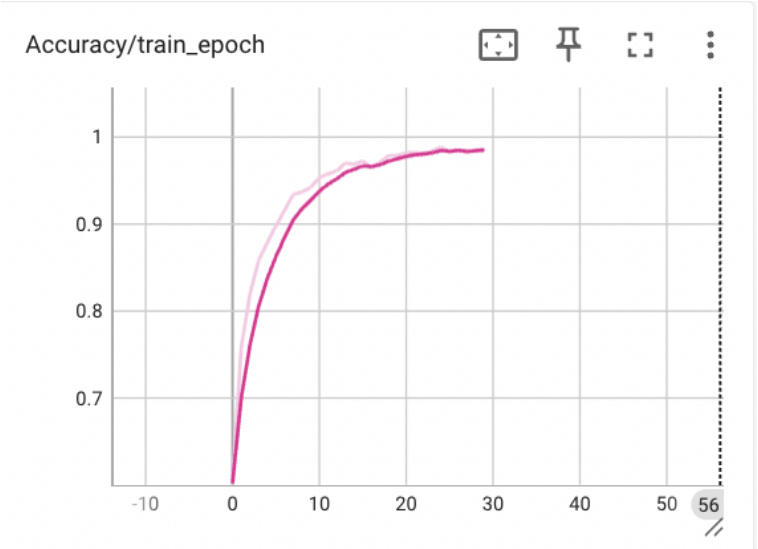

Figure 4.1: Training accuracy for the supervised model on a dataset of beam search solutions for 40 agents in a medium warehouse map. After around 15 epochs, accuracy has quickly risen to above 90%.

4.2.1 Test Accuracy on Branching Decisions

We found that training accuracy quickly rose to around 98% when learning to match expert PBS decisions, seen in Figure. However, this high accuracy was misleading due to the inherent class imbalance in the PBS decision dataset.

Dataset Imbalance: Easy vs. Hard Examples

The PBS decision dataset exhibits a significant imbalance between *easy* and *hard* examples:

Easy Examples comprise the majority of decisions where one child clearly dominates the other according to traditional PBS heuristics. These decisions follow a simple pattern:

- **Clear f-value preference:** One child has significantly lower f-value ($\Delta f > 1$)
- **Tie-breaking by collision count:** When f-values are equal, choose the child with fewer conflicts
- **Predictable outcomes:** Standard heuristic ordering provides the correct answer

Hard Examples represent the challenging cases where heuristic-based decisions are ambiguous or suboptimal:

- **Close f-values:** Children have similar or identical f-values ($\Delta f \leq 1$)
- **Conflicting signals:** Better f-value conflicts with higher collision count
- **Long-term considerations:** Immediate heuristic choice may hurt future search efficiency

The dataset contains approximately 75-85% easy examples, allowing the model to achieve high overall accuracy by simply learning the standard PBS heuristics. The remaining 15-25% of hard examples represent the critical decisions where learned policies could potentially outperform traditional approaches.

Accuracy Breakdown by Example Difficulty

Example Type	Dataset %	Model Accuracy	Baseline DFS Accuracy
Easy Examples	[77.2 %]	[98 %]	[100 %]
Hard Examples	[22.8 %]	[96 %]	[0 %]
Overall	100%	[98%]	[75.15 %]

Table 4.4: Model training accuracy on PBS branching decisions, broken down by example difficulty. Easy examples follow clear heuristic patterns, while hard examples require more sophisticated reasoning. The baseline DFS always follows the heuristic decision.

As seen in Table 4.4, the model is able to achieve very high training accuracy overall in both easy and hard examples. However, the validation dataset results in Table 4.5 show a different story. In validation, model performance on easy and hard cases are completely inversely correlated. We theorize that in training, the model is able to easily learn the heuristic rule and possibly overfits to the training examples, including the hard ones. However, when faced with new agent layout and congestion scenarios, it doesn't generalize as well. As training accuracy increases on hard examples, so does validation accuracy on hard examples, but at the cost of easy example accuracy. Overall, we achieve slightly better validation accuracy on the entire dataset than just running baseline DFS.

Example Type	Dataset %	Model Accuracy	Baseline DFS Accuracy
Easy Examples	[76.8 %]	[81 %]	[100 %]
Hard Examples	[23.2 %]	[58 %]	[0 %]
Overall	100%	[79.3%]	[75.15 %]

Table 4.5: Model validation accuracy on PBS branching decisions, broken down by example difficulty.

4.2.2 Generalization to Unseen Maps

# Agents	Average Tasks			Average Time (s)		
	Model	Normal	Beam Search	Model	Normal	Beam Search
40	725.70	722.73	**767.47**	33.06	**4.87**	31.50
50	804.70	789.65	**889.80**	61.79	**13.44**	78.91
60	807.10	843.40	**926.75**	112.53	**17.55**	251.79

Table 4.6: Performance comparison for 40, 50, and 60 agents between trained SL model, Normal PBS, and Beam Search on the `warehouse_medium.map`. Best values per row are in bold.

Table 4.6 shows the evaluation results of the supervised learning model. Across 30 random seeds, the model almost exactly matches the performance of vanilla DFS, with a slight improvement in higher agent count scenarios, while under performing beam search. An exception is the 60 agent case, when the model does worse than normal DFS. This is mostly expected, given that the model achieves high accuracy on easy decisions consistent with the f-val heuristic in our dataset but low accuracy on decisions that deviate from this, resulting in behavior that closely mirrors the vanilla DFS model. In terms of runtime, the model is slightly faster than beam search but much slower than vanilla DFS.

4.3 Reinforcement Learning PBS Experiments

4.3.1 Training

We trained three RL models for the three different warehouse maps with a different number of agents:

1. Small warehouse map with 25 agents

2. Medium warehouse map with 50 agents

3. Kiva map with 60 agents

These are medium to high congestion scenarios where we believe that RL has the most potential to improve on vanilla PBS. Each simulation was run for 1000 epochs. Because the simulator is not capable of stopping in the middle of PBS search tree, each epoch rollout consists of r solves of full PBS search trees from root to solution node, if found. We want each rollout to include enough consecutive PBS planning steps to capture longer-term dependencies between solutions while not introducing excessive noise. We tried both $r = 4$ and $r = 8$ solves per rollout for 4 random seeds, resulting in 16 and 32 full PBS solve trajectories, respectively. We were unable to support more random seeds per rollout due to lack of parallelization, as discussed previously. We also set the reward horizon parameter to $r/2$ for each case. In the small warehouse scenario, we used a batch size b of 32, an actor learning rate lr_a of 0.0003, and a critic learning rate lr_c of 0.0001. In the medium warehouse and kiva maps, we had $b = 64$, $lr_a = 0.00003$, and $lr_c = 0.00001$. For all scenarios, the decision-level reward weights used were $w_1 = 0.7$ and $w_2 = 0.3$, weighing the number of conflicts resolved and the better f-val, respectively. For the solve-level reward, we used $\alpha = 0.5$ and $\beta = 0.5$. In total episodic reward, we first train the model with $\lambda = 0.2$ as the decision reward weight and $1 - \lambda = 0.8$ as the solve reward weight. The higher heuristic weight helps the model find solution nodes in the beginning. After 400 epochs, we adjust the weights to $\lambda = 0.1$ and $\lambda = 0.9$, again to prioritize throughput over heuristic accuracy. Overall, we wanted the solve reward to be significantly greater in magnitude than any decision-level heuristic reward because it is directly correlated to our evaluation metric of total throughput.

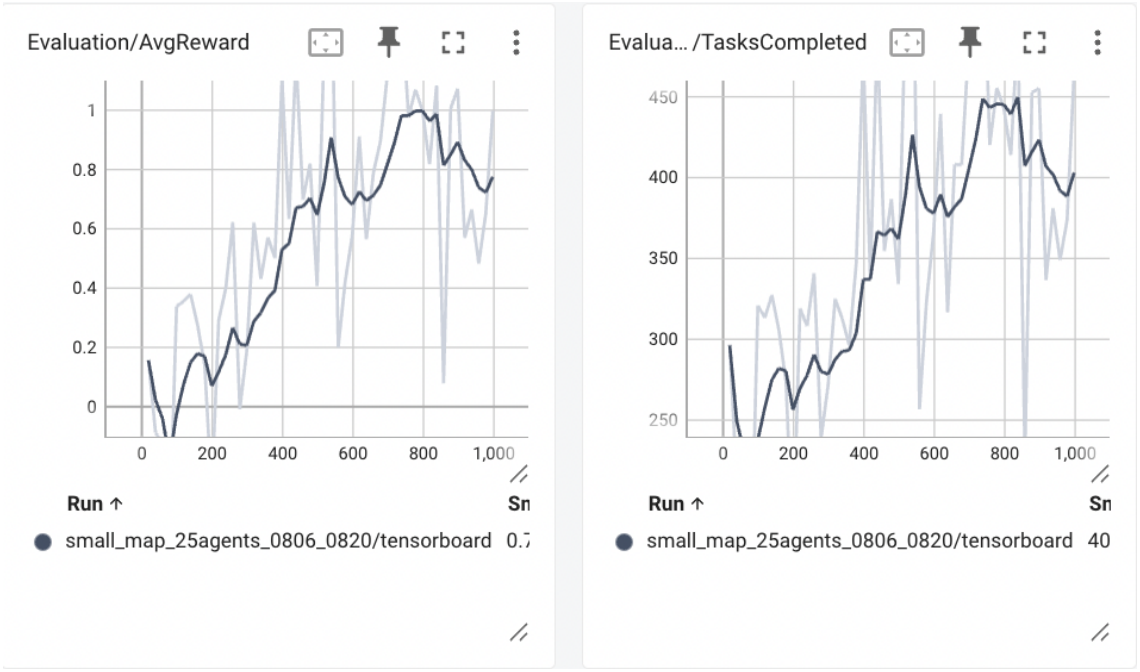

Figure 4.2: Reward and throughput curves for the small warehouse map, 25 agent scenario.

4.3.2 Policy Convergence and Training Curves

Generally, we were able to see both reward and throughput rise consistently throughout training. We found that the training was generally quite unstable for warehouse maps–we attribute part of this to the large distribution shift in agent states as the simulation progress from $T = 0$ to $T = 600$, as agents move from the home and tasking locations across the deck into the aisles. This can be seen in the high-value loss spikes across training, which correspond to when the simulation is reset at $T = 600$ to the next random seed, though increasing the number of random seeds per rollout to 4 helped smooth this out somewhat, as seen in Figure 4.5.

4.3.3 Runtime and Solution Quality

Small warehouse scenario

In the small map, we compare throughput against baselines PBS with an f-val child heuristic, and PBS with a random child heuristic. Our model does not generalize among map layouts

Figure 4.3: Reward and throughput curves for the medium warehouse map, 50 agent scenario. A lot noisier, with a lower overall throughput increase compared to the other maps.

but it is transferable to different numbers of agents, so we test the model (trained on 25 agents) in a 20, 25, and 30 agent scenarios. As seen in Table 4.7, using the model results in significantly higher throughput in all agent densities. Compared to the best performing heuristic model (either DFS or random) in the 20, 25, and 30 agent, scenario, the RL model produced a 47%, 94%, and 53% increase in throughput, respectively. We do see increased runtime, but all 800 step simulations complete in less than a minute, and run faster compared to beam search in the same scenarios.

# Agents	**Average Tasks**			**Average Time (s)**		
	Model	Normal	Random DFS	Model	Normal	Random DFS
20	**462.75**	314.95	310.20	12.33	**2.58**	3.90
25	**488.80**	251.30	346.20	19.76	**5.17**	5.32
30	**440.65**	288.25	332.55	51.61	24.25	**8.09**

Table 4.7: Average tasks completed and average time (seconds) for different numbers of agents across three methods in a small warehouse map. Best value per row is in bold.

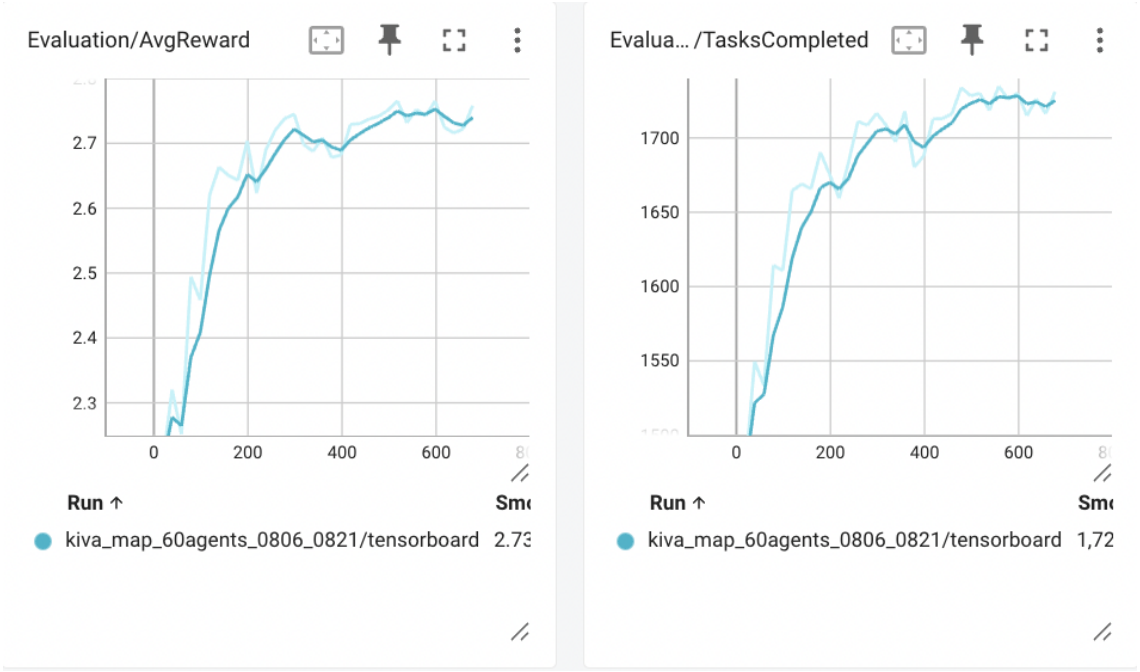

Figure 4.4: Reward and throughput curves for the Kiva map, 60 agent scenario.

# Agents	Average Tasks			Average Time (s)		
	Model	Normal	Random DFS	Model	Normal	Random DFS
40	**683.05**	647.55	652.65	27.48	**5.45**	5.84
50	**782.70**	767.80	765.35	46.10	**10.54**	18.69
60	824.40	**862.00**	811.25	130.10	**17.08**	53.63

Table 4.8: Average tasks completed and average time (seconds) for different numbers of agents across three methods in a medium warehouse map. Best value per row is in bold.

Medium warehouse

In the medium map, we test the trained model (50 agents), in a 40, 50, and 60 agent scenario. Table 4.8 shows that the model provides a less than 5% throughput increase in the 40 and 50 agent scenario, and a 4% decrease in the 60 agent scenario. Again, runtime sees a 7-8x increase.

Kiva

In the Kiva map, the model is a little worse than a normal DFS search- we did not include random DFS as a benchmark because as shown in Table 4.3, the f-val heuristic significantly

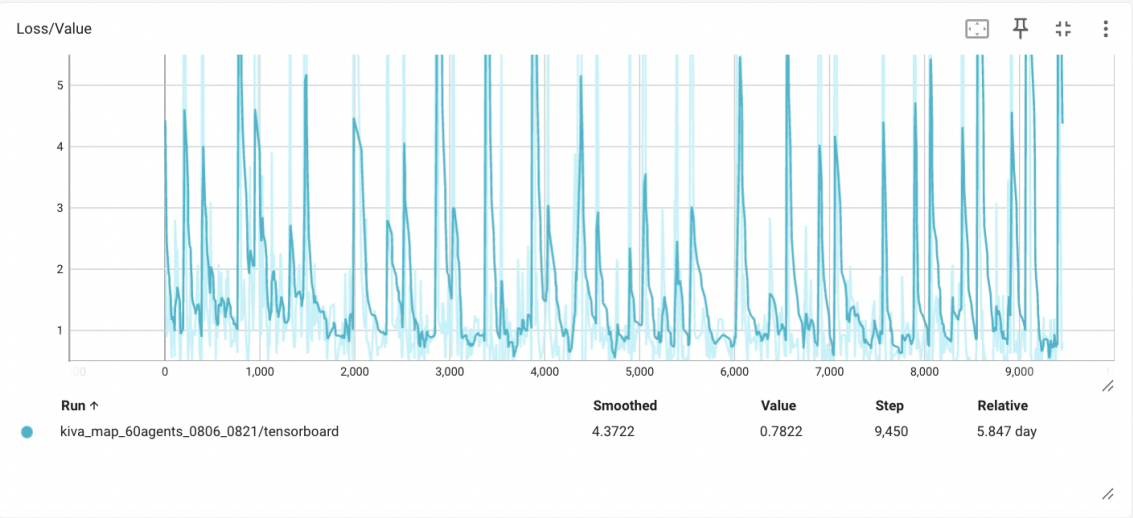

Figure 4.5: Value loss graph for Kiva map. The regular spikes correspond to the resetting of the simulation environment from time 600 to time 0.

# Agents	Average Tasks		Average Time (s)	
	Model	Normal	Model	Normal
60	1718.30	**1769.95**	56.14	**10.05**
80	2116.15	**2199.55**	104.79	**20.29**

Table 4.9: Average tasks completed and average time (seconds) for different numbers of agents across three methods in a Kiva map. Best value per row is in bold.

outperforms random branching. We believe this is because the Kiva map layout, with its regular pod grid and multiple parallel aisles, is less prone to bottlenecks than a more constrained warehouse layout. This means that agents can often take alternative routes without severe slowdown. The extra objective of avoiding congestion may be unnecessary because conflicting agents do not cause bottlenecks the same way as warehouse maps.

4.4 Combined Discussion of Results

Across all experiments, the relative gains from learned or search-augmented PBS methods were strongly dependent on **map structure** and **congestion dynamics**.

In small and medium warehouse maps, we were abe to find an approach that outperformed vanilla PBS with f-val child selection. These environments feature **high obstacle density**

and **frequent choke points**, where naive f-value selection underestimates true traversal time. In such settings, learned or beam search–derived policies could leverage additional context—such as conflict history or potential future bottlenecks—to make better branching decisions, resulting in substantial throughput gains. While beam search achieved larger improvements in the medium map, RL showed better gains on the small map at a lower runtime cost. We believe with more training and a more efficient implementation of our simulation environment, the RL model could match beam search's performance in larger scenarios as well.

In contrast, the Kiva map's open, grid-like structure with multiple redundant paths made it less prone to high-impact congestion. As a result, the static f-val heuristic was already well-aligned with actual execution time, leaving less room for learned or congestion-aware methods to improve performance. In some cases, the extra overhead of the learned policy even slightly reduced throughput compared to vanilla PBS. This highlights that **map topology and congestion risk** are critical factors when choosing between heuristic and learned branching strategies.

From a methodological perspective:

- **Beam search** provides a strong upper bound on achievable decision quality, but its runtime cost scales poorly with agent count and map size.
- **Supervised learning** can imitate beam search effectively on training maps, achieving high decision accuracy, but generalization to lifelong solving in unseen layouts—especially with different congestion patterns may be limited unless the model is adapted to handle a longer-horizon context. Outperforming beam search is difficult due to training dataset imbalances.
- **Reinforcement learning** directly optimizes for throughput without requiring labeled trajectories and can adapt to map-specific dynamics. However, it is more sensitive to reward design, exploration, and state distribution shifts during long simulations.

Summary: For high-congestion, high-density warehouse layouts, learned PBS policies—whether obtained via supervised imitation of beam search or RL—can significantly improve throughput over static heuristics. Beam search provides a strong baseline in decision quality but comes at a high runtime cost, while reinforcement learning shows promising results in smaller maps but requires more tuning for larger scenarios. For low-congestion environments such as Kiva, the marginal benefit of congestion-aware decision-making is small, and traditional heuristics remain a strong choice. For real-world applications, these findings suggest a hybrid deployment strategy: apply learned policies selectively in congestion-prone scenarios while relying on f-value heuristics in open, low-conflict maps.

Chapter 5

Future Work and Conclusion

5.1 Neural Network Improvements

5.1.1 Full Observability

In our simulation, we have full observability of all agent states, paths, and search tree context, but we currently only include node f-val, conflicts, and priorities. With a larger and more efficient architecture, it would be possible to incorporate full path, start/goal, and MDD information as well. Additionally, a recurrent architecture could represent the current tree search context, which may be helpful in making more nuanced decisions.

5.1.2 Parallelization

Our simulator and environment rollout mechanism wasn't capable of parallelization, limiting the variance of the rollout datasets and the generalization ability of the model. Furthermore, any reset of the environment due to too much congestion or a failed solve necessitated starting again at time 0, instead of being able to reset to the state before the solve. Since we are planning in a lifelong context of 800 steps, this may also have caused late-stage congestion scenarios to be underrepresented in the model dataset as well. Adding the ability to roll back the simulation to a previous state instead of resetting could allow the model to quickly learn

to deal with these scenarios. With more time, we would be able to implement these features and hopefully stabilize training.

5.1.3 Imitation Learning

Combining reinforcement learning and imitation learning has shown to lead to more stable training and higher-quality solutions in tree search tasks and robot learning [16], [3]. In the medium map and kiva scenarios, where we know the f-val heuristic outperforms randomness, using the heuristic as an expert while using RL to improve upon policy could be promising.

5.2 Which heuristics to learn?

Our work chose to focus on the modification of only one heuristic in the entire PBS search, based off our initial findings that child selection can cause high variance in solution quality. In the future, it may be interesting to see if learning can choose both the conflict to resolve and the agent priority ordering for that conflict. Even more broadly, we can frame the problem as predicting which edges in the priority tree of a PBS root node need to be added in order to find a high-quality, feasible solution as a one-shot problem rather than sequentially resolving conflicts.

5.3 Conclusion

Overall, our results demonstrate that there is definite value in integrating learning-based decision guidance into traditional PBS solvers for the lifelong MAPF setting. Both supervised learning and reinforcement learning approaches were able to capture patterns in congestion, map layout, and agent interactions that traditional heuristics fail to exploit, leading to improvements in throughput and solution quality in many scenarios.

However, our experiments also reveal that the benefits of learning are not uniform across

all contexts. In less congested layouts, such as Kiva-style maps, traditional heuristics like f-value remain competitive or even superior, highlighting that solver performance is strongly scenario-dependent. These findings emphasize the importance of matching solver strategies to environment characteristics.

By systematically comparing learning-guided PBS variants against established baselines, we provide guidance on which solver paradigms are most effective under varying agent densities, map structures, and congestion profiles. This work not only demonstrates the potential of hybrid approaches but also informs practical deployment strategies for real-world multi-agent systems in warehouse and fulfillment operations.

References

[1] R. Stern et al. "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks." In: *CoRR* abs/1906.08291 (2019). arXiv: 1906.08291. URL: http://arxiv.org/abs/1906.08291.

[2] H. Ma, J. Li, T. S. Kumar, and S. Koenig. "Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks." In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. 2017, pp. 837–845.

[3] G. Sartoretti, J. Kerr, Y. Shi, G. Wagner, T. S. S. Kumar, S. Koenig, and H. Choset. "PRIMAL: Pathfinding via Reinforcement and Imitation Multi-Agent Learning." In: *IEEE Robotics and Automation Letters* 4.3 (2019), pp. 2378–2385.

[4] T. Huang, S. Koenig, and B. Dilkina. "Learning to resolve conflicts for multi-agent path finding with conflict-based search." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 2021, pp. 11246–11253.

[5] J. Yu and S. M. LaValle. "Structure and Intractability of Optimal Multi-Agent Pathfinding." In: *Proceedings of the AAAI Conference on Artificial Intelligence* (2013).

[6] M. Erdmann and T. Lozano-Perez. "On multiple moving objects." In: *Algorithmica* 2 (1987), pp. 477–521.

[7] P. Surynek, A. Felner, R. Stern, and E. Boyarski. "Modifying Optimal SAT-based Approach to Multi-agent Path-finding Problem to Suboptimal Variants." In: *CoRR* abs/1707.00228 (2017). arXiv: 1707.00228. URL: http://arxiv.org/abs/1707.00228.

[8] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant. "Conflict-based search for optimal multi-agent pathfinding." In: *Artificial Intelligence* 219 (2015), pp. 40–66.

[9] H. Ma, D. Harabor, P. J. Stuckey, J. Li, and S. Koenig. "Searching with consistent prioritization for multi-agent path finding." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 7643–7650.

[10] J. Li, A. Tinka, S. Kiesel, J. W. Durham, T. K. S. Kumar, and S. Koenig. "Lifelong Multi-Agent Path Finding in Large-Scale Warehouses." In: *CoRR* abs/2005.07371 (2020). arXiv: 2005.07371. URL: https://arxiv.org/abs/2005.07371.

[11] S. Zhang, J. Li, T. Huang, S. Koenig, and B. Dilkina. "Learning a Priority Ordering for Prioritized Planning in Multi-Agent Path Finding." In: *Proceedings of the Fifteenth International Symposium on Combinatorial Search (SoCS)*. AAAI Press, 2022. URL: https://cdn.aaai.org/ojs/21769/21769-77-25812-1-2-20220717.pdf.

[12] R. Zhou and E. A. Hansen. "Beam-Stack Search: Integrating Backtracking with Beam Search." In: *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS)*. Also available as AAAI Technical Report ICAPS-05-010. AAAI Press, 2005, —. URL: https://cdn.aaai.org/ICAPS/2005/ICAPS05-010.pdf.

[13] M. Phillips and M. Likhachev. "Sipp: Safe interval path planning for dynamic environments." In: *2011 IEEE International Conference on Robotics and Automation*. IEEE. 2011, pp. 5628–5635.

[14] H. Shengyi, R. Antonin, K. Anssi, and W. Weixun. "The 37 Implementation Details of Proximal Policy Optimization." In: *Blog Track at ICLR 2022*. 2022. URL: https://openreview.net/forum?id=Hl6jCqIp2j.

[15] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[16] T. Anthony, Z. Tian, and D. Barber. “Thinking Fast and Slow with Deep Learning and Tree Search.” In: *Advances in Neural Information Processing Systems 30*. Long Beach, CA, USA, 2017.