IMPROVING DECISION-MAKING IN SEARCH ALGORITHMS

FOR COMBINATORIAL OPTIMIZATION WITH MACHINE LEARNING

by

Taoan Huang

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

December 2024

Dedication

To my grandma.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisors, Bistra Dilkina and Sven Koenig, for their unwavering patience and guidance throughout my research journey, for their constructive advice whenever I needed it and for the freedom they offered me to explore diverse research problems. Their enthusiasm, curiosity and dedication to research have profoundly inspired me as a junior researcher.

I am also deeply grateful to the members of my dissertation committee, Lars Lindemann, Meisam Razaviyayn, and Peter Stuckey, for their generous time, insightful feedback and support. A special thanks to Lars for accepting the invitation to join my dissertation committee on short notice.

I would like to extend my heartfelt thanks to Fei Fang, my former undergraduate research advisor, for her pivotal role in shaping my early research career. Fei took me as an intern at Carnegie Mellon University in 2018 when I had no prior research experience. She guided me through my first couple of research projects with extraordinary patience.

I have been fortunate to collaborate with many talented researchers. To Yuandong Tian from Meta AI Research, thank you for many brilliant suggestions for many research problems and our productive collaboration. To Vikas Shivashankar, Michael Caldara and Joseph Durham from Amazon Robotics, thank you for your support and for providing me with the opportunity to work on my first real-world warehouse planning problem. To Jiaoyang Li, thank you for the productive discussions and valuable insights into the multi-agent path finding problem. I would also like to thank my other collaborators, Brandon Amos, Xiaohui Bei, Vadim Bulitko, Junyang Cai, Weizhe Chen, Bohui Fang, Tianyu Gu, Minbiao Han, Weimin

Huang, Thomy Phan, Sumedh Pendurkar, Martin Schubert, Guni Sharon, Weiran Shen, Rohit Singh, Benoit Steiner, Roni Stern, Shuwei Wang, Haifeng Xu, David Zeng, Daochen Zha, Shuyang Zhang and Arman Zharmagambetov, for our fruitful collaborations.

I have thoroughly enjoyed my time with my labmates from the IDM lab and CAIS. To Sina Aghaei, Junyang Cai, Shao-Hung Chan, Weizhe Chen, Aaron Ferber, Amrita Gupta, Weimin Huang, Qing Jin, Caroline Johnston, Nathan Justin, Christopher Leet, Haoming Li, Jiaoyang Li, Laksh Matai, Hannah Murray, Thomy Phan, Caleb Robinson, Qingshi Sun, Bill Tang, Yimin Tang, Yingxiao Ye, Han Zhang and Yi Zheng, thank you for the casual but interesting conversations we had and the many fun lab events.

Finally, I would like to express my deepest gratitude to my parents and my sister for their unconditional love and support. My heartfelt thanks also go to Zixin Huang for her companionship and for being by my side every step of the way.

Table of Contents

List of Tables

List of Figures

Abstract

Combinatorial optimization is fundamental in computer science and operations research, focusing on finding high-quality solutions in structured solution spaces. It encompasses a wide range of real-world problems, including those in logistics, manufacturing, transportation and finance. Many search algorithms have been proposed to solve combinatorial optimization problems (COPs). The high complexity of COPs makes effective decision-making strategies crucial in these algorithms. These strategies guide the search process by navigating the search space. Effective decision-making strategies enhance the efficiency of finding optimal or near-optimal solutions. Despite the significant advancements in search algorithms, human-designed strategies have a few limitations. They often rely on domain-specific knowledge that may not generalize well to different instances and may lead to suboptimal performance. In this dissertation, we show that one can use machine learning (ML) to improve human-designed decision-making strategies for different search algorithms for COPs. Specifically, we focus on two important COPs, namely multi-agent path finding and mixed integer linear programs.

Multi-agent path finding (MAPF) is the problem of finding conflict-free (i.e., collision-free) paths for agents in a shared environment that minimizes their total travel time. It is an NP-hard problem that has important applications for distribution centers, traffic management and computer games. Various search algorithms have been proposed to solve MAPF. Search algorithms, such as Conflict-Based Search (CBS), are guaranteed to find optimal solutions. To trade off runtime with solution quality, bounded-suboptimal search algorithms, such as Enhanced CBS (ECBS), have been proposed to find solutions with a guaranteed

approximation ratio and are more scalable than optimal search algorithms. Unbounded-suboptimal search algorithms, such as Prioritized Planning (PP) and Large Neighborhood Search (LNS), drop optimality guarantees to find solutions even faster. There are a handful of decisions in these search algorithms that concern partitioning the search space, prioritizing search space exploration and pruning the search space. Thus, they typically have a big impact on the efficiency and/or effectiveness of the search. In the past decade of research on MAPF, these decisions have mainly been made manually by humans. In this dissertation, we show that one can leverage ML to improve decision-making in various types of search algorithms for MAPF and introduce CBS+ML, ECBS+ML, MAPF-ML-LNS and PP+ML. Specifically, we apply general ML techniques to learn (1) which conflict to resolve next in CBS, (2) which search tree node to expand next in ECBS, (3) which part of the solutions to improve next in LNS and (4) which priority to assign to agents in PP. In these four settings, we deploy imitation learning to imitate slow but effective experts and reduce the ML task to learning-to-rank problems where the ML models rank available decision options. Empirically, we demonstrate that our ML-guided search algorithms show substantial improvement in terms of the success rates, runtimes and/or solution qualities over their state-of-the-art non-ML-guided counterparts on several different types of grid maps from a popular MAPF benchmark.

Mixed integer linear programs (MILP) are flexible and powerful tools for modeling and solving many difficult real-world COPs. In the past decades, research efforts have been dedicated to improving Branch-and-Bound (BnB), an optimal MILP search algorithm. It is a tree search algorithm that repeatedly breaks the MILP down into smaller subproblems and maintains upper and lower bounds to eliminate subproblems that cannot contain an optimal solution. Unlike BnB, meta-heuristic search algorithms, such as Large Neighborhood Search (LNS) and Predict-and-Search (PaS), are popular unbounded-suboptimal MILP search algorithms that can find high-quality solutions to MILPs much faster without having to prove their optimality. There are important decisions to make in both LNS and PaS: LNS improves the solution by iteratively reoptimizing a subset of variables, and deciding which subset of variables to reoptimize is a challenging

decision to make; PaS predicts which values to assign to a subset of variables based on the input MILP to get a reduced-size MILP that is much faster to solve, and deciding which variables to fix to which values is also important. In previous works, those decisions have mainly been made manually by humans and learned by imitation learning algorithms. In this dissertation, we show that one can leverage contrastive learning to improve decision-making in both LNS and PaS and introduce CL-LNS and ConPaS. In these two settings, we develop novel data collection techniques to collect both positive and negative samples, which are crucial to the success of contrastive learning, and use supervised contrastive losses to train the ML models. Empirically, we demonstrate that both CL-LNS and ConPaS significantly outperform their non-ML-guided and ML-guided counterparts in terms of both runtime and solution quality.

Chapter 1

Introduction

Combinatorial optimization [153] is a pivotal area in computer science and operations research, focusing on selecting the best solutions from vast but structured solution spaces. Such problems are characterized by decisions that are discrete in nature, involving yes/no or select-from-many options. These challenges are prevalent across numerous sectors, including logistics, manufacturing, telecommunications, finance and healthcare, underscoring their significance in both theoretical research and practical applications.

In the past decades, search algorithms for combinatorial optimization problems (COPs) play a crucial role by helping to navigate the solution spaces efficiently and/or effectively. Here, efficiency refers to small runtimes and effectiveness refers to high-quality solutions. On the other hand, machine learning (ML) has profoundly advanced various domains within computer science, revolutionizing traditional approaches and enabling new paradigms of innovation. In this dissertation, we explore leveraging ML to improve search algorithms for COPs and validate the following hypothesis:

One can leverage general ML frameworks to improve decision-making strategies in different types of search algorithms for combinatorial optimization problems.

To explain the hypothesis, we first provide an introduction to COPs in Section 1.1. We then provide an overview of search algorithms for COPs in Section 1.2, where we also talk about decision-making strategies

in search algorithms and what has or has not been done in using ML to improve those strategies. Finally, we present the contribution of this dissertation in Section 1.3.

1.1 Combinatorial Optimization Problems (COPs)

The significance of combinatorial optimization extends across various practical applications. In logistics and transportation, it is employed to optimize routing and scheduling, reducing costs and improving efficiency [189]. In manufacturing, it aids in resource allocation and production scheduling [165, 109]. In network design and management, it is applied to optimizing the operation of transportation and utility networks to enhance the networks' throughput or reliability [84, 67].

The primary objective in COPs is to optimize a particular objective function, such as cost, time or resource usage, subject to a set of constraints. This could involve, for example, finding the most cost-effective route in a delivery network that adheres to constraints such as distance and capacity limits. Problems in this field are inherently complex, often requiring sophisticated mathematical models and algorithms. The complexity arises from the exponential growth of possible solutions as the instance size increases, making many of these problems NP-hard and thus challenging to solve [153].

In this dissertation, we focus on two specific NP-hard COPs, namely multi-agent path finding (MAPF) and mixed integer linear programs (MILPs). Next, we introduce both MAPF and MILP.

1.1.1 Multi-Agent Path Finding (MAPF)

MAPF is a COP that involves finding the optimal paths for multiple agents to navigate from their respective start locations to their target locations without colliding with each other and static obstacles in a potentially congested shared environment. The objective of MAPF typically involves minimizing the sum of all agents' travel times.

We focus on MAPF in this dissertation since it has significant relevance across multiple real-world domains: In robotics, it is crucial to coordinate the movement of autonomous robots in warehouses or factories, where efficient path planning can drastically improve effectiveness and safety [199]. In computer games, MAPF algorithms enhance the realism by managing the non-player characters to move in certain formations [140, 125] which contributes to more enjoyable gameplay. In transportation management and logistics, MAPF algorithms benefit applications like drone delivery systems [81] and autonomous vehicle coordination [45, 123, 33], which also help to reduce traffic congestion and improve safety. In disaster response scenarios, managing the paths of multiple agents efficiently is also pivotal since search and rescue operations rely on coordinated movements of responders and evacuees to maximize coverage and minimize response times [161, 157]. Overall, the significance of MAPF lies in its ability to enhance efficiency and effectiveness in systems involving multiple agents that are not allowed to collide. As the deployment of multi-agent systems continues to grow, the ability to solve MAPF problems efficiently and effectively becomes increasingly critical.

However, solving MAPF optimal is an NP-hard problem on general graphs [208] and some specific types of graphs, such as planar graphs [207] and grid graphs [9], meaning that finding the optimal solution can require computational time that grows exponentially with the problem instance size. Even approximate solutions can be challenging to obtain within a reasonable runtime [139]. The complexity arises from the need to account for the interactions between agents, which leads to exponential growth in the search space as the number of agents increases. This difficulty is even exacerbated in dynamic environments where agents must adapt to changing conditions in real-time [126] or when path planning is coupled with other tasks such as target assignments [138].

1.1.2 Mixed Integer Linear Program (MILP)

In addition to MAPF, we also focus on mixed integer linear programs (MILP) since they are powerful mathematical modeling techniques, and a wide range of COPs can be formulated as MILPs. MILPs extend linear programs by allowing some or all of the decision variables to be restricted to integer values. The general form of a MILP involves an objective function, which is to be maximized or minimized, subject to a set of linear equality and/or inequality constraints. Including integer variables in MILPs makes them particularly useful for solving problems requiring discrete decisions. MILPs have applications in numerous domains due to their flexibility in modeling complex decision-making: In operations research, MILPs are used to solve scheduling [109, 146], resource allocation [165] and supply chain management problems [204, 93, 171]. In finance, they are applied to portfolio optimization [13, 145], where investment decisions must comply with risk control and budget constraints. In the energy sector, MILPs are used to optimize power generation and distribution [29, 152, 129]. In logistics and transportation, MILPs help in vehicle routing and network design [134, 65]. The versatility of MILPs makes them an essential tool for industries where optimal decision-making is crucial.

However, MILPs are generally NP-hard to solve. This complexity arises from the integer constraints, which create a combinatorial explosion of possible solutions. Unlike linear programs, where efficient polynomial-time algorithms exist, large-scale MILP instances with numerous variables and constraints are much harder to solve, requiring significant computational resources and sophisticated algorithms.

1.2 Search Algorithms for COPs

Search algorithms for COPs have been developed and play a crucial role by helping to navigate the solution spaces efficiently and effectively. To name a few, there are optimal search algorithms, such as A* search [72] and Branch-and-Bound (BnB) [113], bounded-suboptimal search algorithms, such as focal search [155]

and explicit estimation search [186], and unbounded-suboptimal search algorithms, such as local search [97] and genetic algorithms [79]. Many solvers for COPs based on these search algorithms have evolved considerably over the past decades and can handle large-scale problem instances thanks to algorithmic advances. There are many decisions to make in search algorithms that are crucial to their success. These decisions concern the exploration and exploitation of the search space. These decisions include partitioning the search space into multiple parts, determining the order in which to explore these parts and deciding which part of the search space should be pruned.

In the context of MAPF, recent research has focused on developing efficient MAPF search algorithms to tackle its inherent complexity. MAPF search algorithms can be broadly categorized into centralized and decentralized methods. Centralized methods, such as Conflict-Based Search (CBS) [174] and its variants [10, 120, 124], provide optimal or bounded-suboptimal solutions by considering the joint state space of all agents that includes information of all agents' locations at every time step. Decentralized methods [78, 172], on the other hand, allow agents to plan independently while coordinating with other agents to avoid conflicts (i.e., collisions), offering scalability benefits. However, decentralized methods typically find lower-quality solutions than centralized methods. Thus, we focus on centralized methods in this dissertation.

Research on MILPs is a dynamic and evolving field, with ongoing efforts to develop more efficient algorithms. In the past decades, innovations in optimal MILP search algorithms, such as BnB [113] and Branch-and-Cut [64], have significantly improved the efficiency of solving MILPs and become the cores of many state-of-the-art solvers, such as SCIP [21], Gurobi [69] and CPLEX [37]. On the other hand, popular unbounded-suboptimal search algorithms, such as greedy algorithms [27] and local search [114, 71, 177], provide solutions faster and are often good enough for practical purposes. Other unbounded-suboptimal search algorithms, such as genetic algorithms [136], simulated annealing [185] and particle swarm optimization [105], have also been explored to solve MILP problems.

There are notable similarities between some MAPF and MILP search algorithms. For example, CBS in MAPF and BnB in MILP are both tree search algorithms that repeatedly break problems down into smaller subproblems. CBS selects conflicts to resolve, while BnB selects variables to branch on. These decisions significantly impact the efficiency of the search algorithms [102, 22]. Another important decision shared by both BnB and the unbounded suboptimal variants of CBS* is selecting which search tree nodes to expand. Large Neighborhood Search (LNS) is another search algorithm applied to both MAPF and MILP. LNS begins with a feasible solution and iteratively improves it by reoptimizing a part of the solution. For both MAPF and MILP, selecting which part of the solution to reoptimize is a crucial decision in LNS [117, 177, 198, 179].

The idea of combining research efforts in MAPF and MILP search algorithms has been explored, particularly in formulating MAPF as a MILP. [209] models MAPF as a multi-commodity flow problem using MILPs. This approach involves a time-expanded graph where vertices are indexed by location and time, with binary variables for each pair of agents and edges in the graph. For small MAPF instances, these MILPs can be solved using off-the-shelf MILP solvers. However, this formulation does not scale well to large instances due to its inefficient representation of MAPF. The Branch-and-Cut-and-Price algorithm [112] is a MILP-based MAPF search algorithm that addresses this issue by incrementally constructing the necessary variables and constraints to resolve conflicts in agents' paths.

1.2.1 Machine Learning (ML) for COPs

Despite the advances in search algorithms and their ability to tackle large-scale COPs, many state-of-the-art search algorithms still rely on hand-crafted strategies to make decisions that are otherwise too expensive to compute or not well-defined mathematically. These hand-crafted strategies inherently face several limitations since they often rely on domain-specific knowledge and intuition, which, while valuable, can

---

*BnB maintains both upper and lower bounds on the solution. Thus, the order in which search tree nodes are expanded does not affect the proof of optimality. CBS and its variants maintain only the lower bounds. Thus, CBS has little flexibility in selecting search tree nodes since it always selects the ones with the lowest bounds, but its bounded-suboptimal variants do not.

```
┌──────────────┐         ┌──────────────────┐         ┌──────────┐
│ An ML Method │────────▶│ A Search Algorithm│────────▶│  A COP   │
└──────────────┘         └──────────────────┘         └──────────┘
```
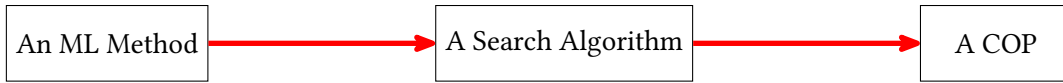
Figure 1.1: An ML method that applies to a search algorithm for a COP.

```
                                                      ┌──────────┐
                                                      │  COP 1   │
                                                      └──────────┘
┌──────────────┐         ┌──────────────────┐         ┌──────────┐
│ An ML Method │────────▶│ A Search Algorithm│────────▶│  COP 2   │
└──────────────┘         └──────────────────┘         └──────────┘
                                                      ┌──────────┐
                                                      │  COP 3   │
                                                      └──────────┘
```
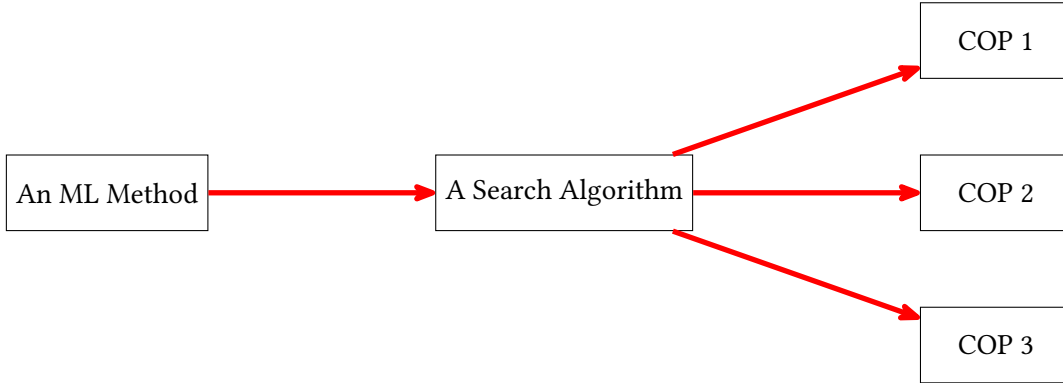
Figure 1.2: An ML method that applies to a search algorithm for multiple COPs.

be subjective and may not generalize well across different problem instances or problem sizes. Moreover, manually tuning parameters and designing effective strategies for balancing exploration and exploitation in the search space is a complex and time-consuming task, prone to human errors and bias. This can lead to suboptimal performance, especially in new environments where generalizable decision-making is crucial.

Given these limitations, ML presents a promising avenue for enhancing decision-making in search algorithms for COPs [14]. ML can learn from vast amounts of data, identifying patterns and strategies that may not be apparent to human designers. By leveraging techniques such as supervised learning and imitation learning, it is possible to develop decision-making models that make decisions more efficiently and effectively. These models can dynamically adjust parameters, balance exploration and exploitation, and make decisions based on the state of the search that includes, for example, information on the solutions found and statistics on the search process. Consequently, integrating ML into combinatorial optimization has the potential to significantly improve the efficiency and effectiveness of solving complex real-world problems.

ML has been applied to improve search algorithms for various COPs. Much of the previous research has focused on developing ML methods tailored to specific problems, including routing problems such as

the traveling salesman problem (TSP) [12, 42] and the vehicle routing problem (VRP) [149, 107, 132, 127], as well as resource allocation problems such as the bin packing problem [83, 47] and the scheduling problem [108, 215]. These studies specialize in one COP and propose ML methods tailored to one search algorithm, as illustrated by Figure 1.1. However, this narrow focus presents challenges in generalization: ML methods developed for a particular search algorithm in one COP cannot easily be adapted to other search algorithms for the same COP or the same search algorithm for different COPs. For example, applying ML methods designed for local search in VRP [132, 127] to greedy algorithms for VRP [12, 107] is not straightforward, as these methods use different features, network architectures and learning algorithms. Similarly, ML methods for local search in VRP may not be easily transferable to local search for other COPs, as they are often tailored to the specific features and structures of VRP.

There are studies that partially address this issue by developing an ML method for a specific search algorithm that can be applied to multiple COPs, as illustrated by Figure 1.2. For example, one of the earliest works on ML-guided COP [99] improves a greedy algorithm to solve multiple graph optimization problems, including TSP, the maximum cut problem and the maximum independent set problem; [28] learns to improve a search algorithm based on constraint programming and dynamic programming for TSP, the portfolio optimization problem and the packing problem; [32] learns to perform reoptimization in local search for expression simplification, the scheduling problem and VRP. However, these approaches are limited by their focus on specific search algorithms, restricting their applicability to only those COPs that the search algorithms can address. For example, the greedy algorithm in [99, 143] constructs solutions by sequentially adding vertices, which works well for graph optimization problems where feasible solutions are easy to obtain but may not generalize to other problems with more complex constraints.

MILP itself is a COP and serves as a general technique for solving various COPs, partially addressing the limitations of earlier studies by modeling and solving a broader range of problems. Recent research has integrated ML with MILP search algorithms to enhance decision-making strategies. These efforts
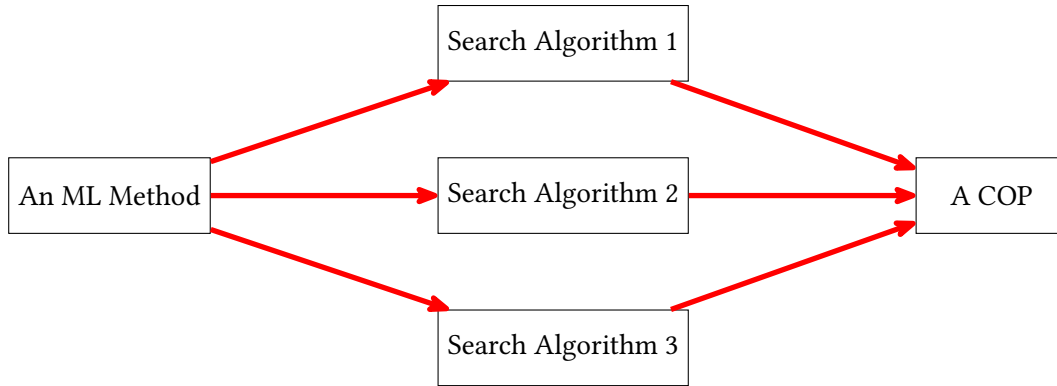
Figure 1.3: An ML framework: An ML method that applies to multiple search algorithms for a COP.

are motivated by the observation that MILP instances in certain applications often share structural similarities [6]. A growing body of literature has emerged to enhance MILP search algorithms, particularly Branch-and-Bound [102, 59] and Large Neighborhood Search [177, 198, 179], by incorporating adaptable ML components that leverage data and training.

In the context of MAPF, there has been progress as well in integrating ML techniques to enhance decentralized MAPF algorithms. For instance, reinforcement learning and imitation learning have been applied to construct agents' paths in warehouse environments with limited communication capabilities [172, 38]. However, to the best of our knowledge, there has been little progress in advancing centralized MAPF search algorithms with ML.

Several gaps remain in the current literature:

1. **Lack of ML methods for Centralized MAPF Search Algorithms** Despite the similarities in decision-making between MAPF and MILP search algorithms, there has been little progress on applying ML to improve decision-making in centralized MAPF search algorithms. Leveraging insights from ML-guided MILP algorithms is a promising research direction to address this gap.

2. **Limited Exploration of ML Techniques** Most studies in ML for COPs have focused on imitation learning and reinforcement learning, leaving opportunities for other emerging ML techniques. For example, contrastive learning, which has seen success in computer vision [77, 74, 30], natural

language processing [62, 164] and graph representation learning [205, 188], could be explored to improve decision-making strategies in search algorithms for COPs.

3. **Need for General ML Frameworks** Most crucially, a significant gap is the absence of general ML frameworks that can be applied to multiple search algorithms for a specific COP, as illustrated in Figure 1.3. While previous studies have developed ML methods for a specific search algorithm for one or multiple COPs, there is no existing work that formulates a general ML framework capable of improving decision-making strategies for multiple search algorithms for a single COP. This is important because (1) different types of search algorithms are needed since the desired trade-offs between optimality requirements of the solutions and the computation budgets to solve a COP change when solving different instances under different circumstances; (2) valuable common insights into a COP and engineering techniques can be utilized to improve multiple search algorithms; and (3) a general ML framework could enable the reuse of ML implementations, making tools more accessible to users, including those without extensive ML expertise. Developing such a framework is challenging, as it requires not just an ML method but a general ML framework that can be applied to diverse search algorithms designed to function differently.

1.3   Contributions

In this dissertation, we fill the gaps in the current literature and make two major contributions. The first major contribution validates the hypothesis for MAPF, addressing the first and third gaps. The second major contribution validates the hypothesis for MILP, addressing the second and third gaps. Addressing the second gap for MAPF is left for future work and discussed in Chapter 4.

- **Contribution 1** To validate the hypothesis for MAPF, we first formulate a general imitation-learning framework to improve decision-making strategies for MAPF search algorithms. Building
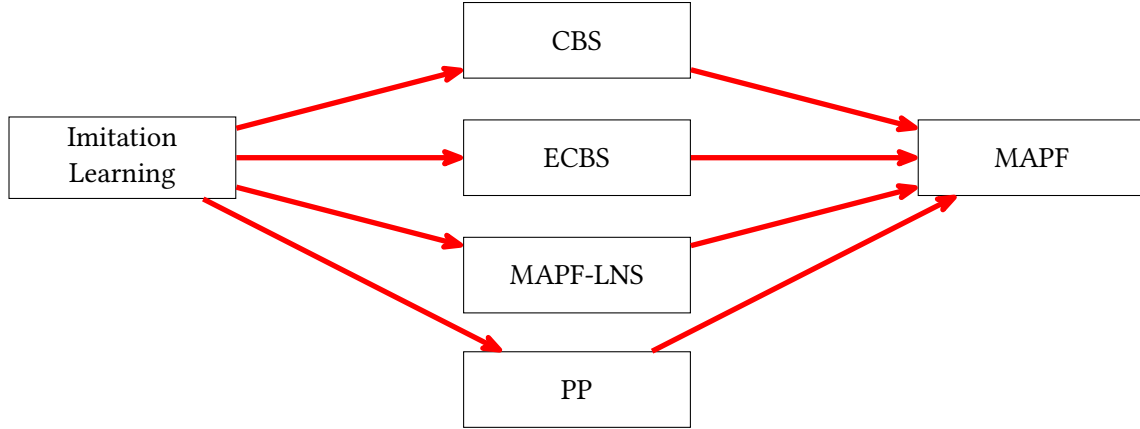
Figure 1.4: Contribution 1: An ML framework that applies imitation learning to multiple MAPF search algorithms.
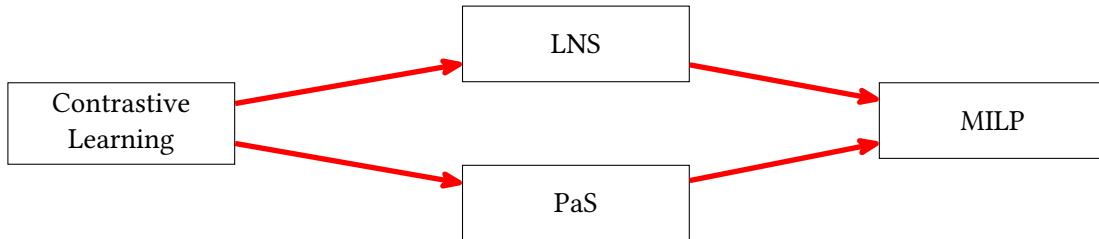
Figure 1.5: Contribution 2: An ML framework that applies contrastive learning to multiple MILP search algorithms.

on existing imitation learning methods for MILP solving [102, 73], we tailor our framework specifically for MAPF. We utilize domain knowledge from MAPF to design the data collection processes and features, which are crucial for imitation learning. We then apply this framework to four different state-of-the-art MAPF search algorithms, namely (1) Conflict-Based Search (CBS), (2) Enhanced Conflict-Based Search (ECBS), (3) Large Neighborhood Search (MAPF-LNS) and (4) Prioritized Planning (PP), as illustrated in Figure 1.4. CBS is an optimal tree search algorithm that repeatedly detects conflicts between agents and resolves one of them by splitting the current problem into two subproblems. For CBS, we propose CBS+ML, that learns to select which conflict to resolve. ECBS is a bounded-suboptimal variant of CBS that expands the search tree by repeatedly selecting search tree nodes from a list of candidate nodes. For ECBS, we propose ECBS+ML, that learns to select which search tree node to expand next. MAPF-LNS is an anytime MAPF search algorithm that iteratively

selects a subset of agents' paths to reoptimize. For MAPF-LNS, we propose MAPF-ML-LNS, that learns to select promising subsets. PP is a greedy search algorithm that plans agents' paths sequentially in descending order of their preassigned priorities. For PP, we propose PP+ML, that learns to assign priorities to agents. Finally, we empirically show that CBS+ML, ECBS+ML, MAPF-ML-LNS and PP+ML significantly outperform their respective vanilla versions in terms of runtime and/or solution quality. Further details are provided in Chapter 2.

- **Contribution 2** To validate the hypothesis for MILP, we first formulate a general contrastive-learning framework to improve decision-making strategies for MILP search algorithms. We then apply this framework to two different state-of-the-art MILP search algorithms as illustrated in Figure 1.5, namely (1) Large Neighborhood Search (LNS) and (2) Predict-and-Search (PaS). LNS is an anytime MILP search algorithm that iteratively selects a subset of variables to reoptimize. For LNS, we propose CL-LNS, that learns to select promising subsets. PaS is a greedy search algorithm that first greedily fixes values for a subset of variables and then solves a reduced-size MILP. For PaS, we propose ConPaS, that learns to predict which values to fix for which subset of variables. Finally, we empirically show that both CL-LNS and ConPaS significantly outperform their respective ML-guided and non-ML-guided counterparts in terms of both runtime and solution quality. Futher details are provided in Chapter 3.

To summarize, we introduce two general ML frameworks aimed at improving human-designed decision-making strategies in different search algorithms for MAPF and MILP, respectively. They are the first ML frameworks capable of improving multiple search algorithms for a single COP. Although MAPF cannot be solved efficiently when using its MILP formulation, as discussed in Section 1.2, we leverage insights from ML methods tailored for MILP search algorithms to formulate an ML framework that improves four different MAPF search algorithms. They are the first works that use ML techniques to enhance MAPF search

algorithms and the first ML framework that provides systematic guidance on improving these algorithms with ML.

For MILP, previous research has applied imitation learning and reinforcement learning to improve decision-making strategies in various MILP search algorithms. However, no prior work has demonstrated how to systematically apply a general ML framework across multiple search algorithms, nor has any used contrastive learning, an ML technique proven more effective than imitation learning and reinforcement learning in other domains. We address these gaps by proposing the first contrastive learning-based ML framework for MILP and applying it to improve two MILP search algorithms.

Finally, we suggest that the contrastive learning framework developed for MILP can also be generalized to MAPF search algorithms. Moreover, we propose that both ML frameworks could be generalized to other COPs within the context of multi-task learning. These possibilities, along with the realized impact of our two major contributions, are discussed in Chapter 4.

MAPF	Multi-Agent Path Finding
MILP	Mixed Integer Linear Program
COP	Combinatorial Optimization Problem
ML	Machine Learning
CBS	Conflict-Based Search
ECBS	Enhanced Conflict-Based Search
LNS	Large Neighborhood Search
PP	Prioritized Planning
BnB	Branch-and-Bound
PaS	Predict-and-Search
TSP	Traveling Salesman Problem
VRP	Vehicle Routing Problem
CT	Constraint Tree
PPS	Parallel-Push-and-Swap
WDG	Weighted Dependency Graph
ICBS	Improved Conflict-Based Search
MDD	Multi-Valued Decision Diagram
CG	Conflict Graph
EECBS	Explicit Estimation Conflict-Based Search
SVM	Support Vector Machine
LB	Local Branching
ND	Neural Diving
CL	Contrastive Learning
LP	Linear Program
GAT	Graph Attention Network
PG	Primal Gap
PI	Primal Integral
DNN	Deep Neural Network
AUC	Area Under the Curve
GCN	Graph Convolutional Network
MLP	Multi-Layer Perceptron
MVC	Minimum Vertex Cover
MIS	Maximum Independent Set
CA	Combinatorial Auction
SC	Set Covering
IP	Item Placement

Table 1.1: Acronyms and their meanings.

Chapter 2

Improving Decision-Making in MAPF Search Algorithms

In this chapter, we present the first major contribution of this dissertation. Specifically, we formulate a general imitation learning framework to improve decision-making strategies for MAPF search algorithms. We identify important decisions that are typically made by human-designed strategies in four different state-of-the-art MAPF search algorithms, namely Conflict-Based Search (CBS), Enhanced Conflict-Based Search (ECBS), Large Neighborhood Search (MAPF-LNS) and Prioritized Planning (PP), and then apply the framework to improve them. Empirically, the machine learning-guided versions of the MAPF search algorithms substantially outperform their non-ML-guided counterparts in terms of runtime and/or solution quality. Therefore, these results validate the hypothesis that one can leverage a general ML framework to improve human-designed decision-making strategies in different types of MAPF search algorithms.

The remainder of this chapter is structured as follows. In Section 2.1, we state the motivation behind using machine learning (ML) for MAPF and provide an overview of our contributions. In Section 2.2, we formally define MAPF. In Section 2.3, we introduce CBS, ECBS, MAPF-LNS and PP. In Section 2.4, we summarize related work. In Section 2.5, we introduce the framework. In Sections 2.6-2.9, we introduce CBS+ML, ECBS+ML, MAPF-ML-LNS and PP+ML, respectively, and evaluate them empirically. Finally, in Section 2.10, we summarize the contributions of this chapter.

2.1 Introduction

Multi-Agent Path Finding (MAPF) is the problem of finding a set of conflict-free (that is, collision-free) paths for a team of agents that moves on a given underlying graph and minimizes the sum of path costs. MAPF has practical applications in distribution centers [138, 80], traffic management [45] and video games [140]. For these applications, a MAPF instance can involve hundreds and sometimes thousands of agents.

MAPF is NP-hard to solve optimally [209, 9]. However, given its importance in various applications, different MAPF search algorithms have been proposed. One of the leading categories of MAPF search algorithms is optimal and bounded-suboptimal algorithms, which guarantee to return a solution that is optimal and not larger than optimal by more than some user-specified multiplicative factor $w \geq 1$, respectively. The state of the art in this category includes Conflict-Based Search (CBS) [174], Branch-and-Cut-and-Price [112], Enhanced CBS (ECBS) [10] and Explicit Estimation CBS (EECBS) [124]. CBS is an optimal MAPF search algorithm and the backbone of most of these algorithms. It uses a single-agent path-finding algorithm to plan a path for each agent first and resolves conflicts afterward. The key idea behind CBS is to use a bi-level search that resolves conflicts by adding constraints at the high level and replans paths for agents respecting these constraints at the low level. The high level of CBS performs a best-first search on a binary search tree called *constraint tree* (CT). A CT node consists of a set of paths, one for each agent, and a set of constraints on these paths. The cost of a CT node is the sum of costs of all agents' paths. CBS maintains an open list that sorts all CT nodes that have not been expanded in increasing order of their costs. CBS always expands the first CT node in the open list. To expand a CT node, CBS chooses a conflict between two agents' paths to resolve and adds constraints that prevent this conflict in the child CT nodes. ECBS is a bounded-suboptimal version of CBS. It uses focal searches [155] instead of best-first searches for both the high-level and the low-level searches to guarantee bounded suboptimality. The high-level search of ECBS maintains a focal list that contains the subset of CT nodes in the open list whose costs are at most

w times the lowest cost of any CT node in the open list and can select an arbitrary one in the focal list for expansion.

The other category of MAPF search algorithms is unbounded-suboptimal algorithms that can solve very large MAPF instances but usually find low-quality solutions. These algorithms include greedy algorithms, such as prioritized planning (PP) [49], rule-based algorithms, such as Parallel-Push-and-Swap (PPS) [170] and Priority Inheritance with Backtracking [150], and anytime algorithms*, such as Large Neighborhood Search for MAPF (MAPF-LNS) [117]. MAPF-LNS is one of the state-of-the-art MAPF search algorithms in this category. MAPF-LNS first finds a set of conflict-free paths quickly using an existing MAPF search algorithm, such as EECBS, PP or PPS. It then improves the sum of costs of the conflict-free paths to near-optimal over time by iteratively destroying subsets of the paths generated by *agent-set selection strategies* and replanning them using a *repair operator* while leaving the remaining paths unchanged. On the other hand, PP is one of the fastest algorithms for solving MAPF suboptimally based on a simple planning scheme [49] that assigns each agent a unique priority and computes, in descending priority ordering, each agent's individually cost-minimal path that avoids conflicts with both static obstacles and the paths of the already-planned agents (which are treated as moving obstacles).

There is a lot of important decision-making in MAPF search algorithms that concerns, for example, how to partition the search space into two or more parts, which part of the search space to explore next and how to prune the search space. In the past, decision-making was often done by hand-crafted strategies which requires domain knowledge and a thorough understanding of the algorithms. In this chapter, we apply a general ML framework to learn such strategies and demonstrate that the performance of MAPF search algorithms, i.e., their runtime and/or solution quality, can be improved with ML-guided strategies. In particular, we introduce CBS+ML, ECBS+ML, MAPF-ML-LNS and PP+ML to show that the framework is applicable to state-of-the-art MAPF search algorithms of different types: the optimal algorithm CBS with

---

*An anytime algorithm can be stopped at any point after a feasible solution is found during its execution and still provides a valid solution to the problem.

improved heuristics [120], the bounded-suboptimal algorithm ECBS [10] and the unbounded-suboptimal algorithms MAPF-LNS [117] and PP [49]. To apply this ML framework to improve a MAPF search algorithm, we first identify an important decision to make in the search. For example, there are strategies that decide which conflict to resolve next in CBS, which node in the focal list to expand next in ECBS, which subset of agents to replan next in MAPF-LNS and which priorities to assign to agents in PP. We then learn to imitate effective decisions from an expert: for CBS, we propose an expert to select the next conflict to resolve based on a weighted dependency graph (WDG) heuristic [120] that solves a weighted vertex cover problem for each conflict on a graph that captures interaction among agents; for ECBS, the expert selects the next node to expand by retrospectively computing the complete search tree; for MAPF-LNS, the expert samples agent subsets and replans them to select the best one; for PP, the expert samples random priority orderings and plans agents' paths with each of them to select the best one. These experts are too slow to be directly useful in the search but provide effective guidance for the search. By observing and recording the features and decisions of the expert, we deploy imitation learning to learn strategies to predict decisions that are as similar as possible to the expert without actual exhaustive computation. Empirically, we show that variants of MAPF search algorithms with the learned strategies substantially outperform their non-ML-guided counterparts in terms of runtime and/or solution quality. The results demonstrate how a general ML framework can be applied to advance some state-of-the-art MAPF search algorithms and possibly others.

2.2 Multi-Agent Path Finding

The *Multi-Agent Path-Finding (MAPF) problem* is to find a set of conflict-free paths for a set of agents $\{a_1, \ldots, a_k\}$ on a given 2D four-neighbor grid map with blocked cells that is represented as an undirected unweighted graph $G = (V, E)$. Each agent a_i has a start vertex $s_i \in V$ and a goal vertex $t_i \in V$. A path $p_i = (p_{i,0}, \ldots, p_{i,l(p_i)})$ for agent a_i is a sequence of vertices, where $p_{i,0} = s_i$, $p_{i,l(p_i)} = t_i$ and $l(p_i)$

is the length of the path. Time is discretized into time steps, and, at each time step t, every agent takes an action: It either moves to an adjacent vertex, i.e., $(p_{i,t}, p_{i,t+1}) \in E$, or waits at its current vertex, i.e., $p_{i,t} = p_{i,t+1} \in V$. Two types of conflicts are considered: i) A vertex conflict $\langle a_i, a_j, v, t \rangle$ occurs when agents a_i and a_j are at the same vertex v at time step t; and ii) an edge conflict $\langle a_i, a_j, u, v, t \rangle$ occurs when agents a_i and a_j traverse the same edge (u, v) in opposite directions from time step t to time step $t + 1$. The cost of agent a_i is defined as $l(p_i)$, which is the number of time steps until it reaches its goal vertex t_i and remains there. The delay of agent a_i is defined as the difference between $l(p_i)$ and the distance between its start and goal vertices. A solution is a set of conflict-free paths that move all agents from their start vertices to their goal vertices. The sum of costs (and delays) of a solution is the sum of all agent costs $\sum_{i=1}^{k} l(p_i)$ (and their delays, respectively). Our goal is to find a solution with the minimum sum of costs.

2.3 Background

In this section, we provide a brief introduction to MAPF search algorithms, CBS, ECBS, MAPF-LNS and PP, that we focus on in this chapter. At the end, we introduce the MAPF instances used in the empirical evaluation.

2.3.1 Conflict-Based Search (CBS)

CBS is a bi-level tree search algorithm. It records the following information for each CT node N:

1. N_{Con}: The set of constraints imposed so far in the search. There are two types of constraints: i) a vertex constraint $\langle a_i, v, t \rangle$, corresponding to a vertex conflict, prohibits agent a_i from being at vertex v at time step t; and ii) an edge constraint $\langle a_i, u, v, t \rangle$, corresponding to an edge conflict, prohibits agent a_i from moving from vertex u to vertex v between time steps t and $t + 1$.

2. N_{Sol}: A set of individually cost-minimal paths for all agents respecting the constraints in N_{Con} that are potentially not conflict-free. An individually cost-minimal path for an agent is a cost-minimal path between its start and goal vertices under the assumption that it is the only agent in the graph.

3. N_{Cost}: The cost of N, calculated as the sum of costs of all agents in N_{Sol}.

4. N_{Conf}: The set of conflicts between any two paths in N_{Sol}.

On the high level, CBS starts with a CT with only one CT node whose set of constraints is empty and then expands the CT in a best-first manner by always expanding a CT node with the lowest N_{Cost}. After choosing a CT node to expand, CBS identifies the set of conflicts N_{Conf} in N_{Sol}. If there are none, CBS terminates and returns N_{Sol}. Otherwise, CBS randomly (by default) selects one of the conflicts to resolve and adds two child CT nodes to N by imposing, depending on the type of conflict, an edge or vertex constraint on one of the two conflicting agents and adding the constraint to N_{Con} of one of the child nodes and similarly for the other conflicting agent and N_{Con} of the other child node. On the low level, it replans the paths in N_{Sol} to accommodate the newly-added constraints, if necessary. CBS guarantees optimality by performing best-first searches on both of its high and low levels. CBS itself does not guarantee completeness, but [210] has an algorithm to detect whether a solution exists for a MAPF instance which can be run to guarantee completeness prior to running CBS. We do not implement such a component in our empirical evaluation.

2.3.2 Enhanced CBS (ECBS)

ECBS is a bounded-suboptimal version of CBS [10] and is summarized in Algorithm 1. Given a suboptimality factor $w \geq 1$ (Line 1), ECBS is guaranteed to find a w-approximate solution. Both the high-level and low-level searches of ECBS use focal searches [155] instead of best-first searches. Consider a CT node N. On the low level, ECBS runs a focal search for each agent a_i (Line 12) such that the cost of the path found is at most $wN_{\mathsf{LB},i}$, where $N_{\mathsf{LB},i}$ is the lower bound on the cost of the individually cost-minimal

---
Algorithm 1 ECBS

  1: **Input:** A MAPF instance and suboptimality factor w

  2: Generate the root CT node R and calculate R_{Sol}, R_{Cost} and R_{Conf}

  3: Initialize open list $\mathcal{N} \leftarrow \{R\}$

  4: $\mathsf{LB} \leftarrow R_{\mathsf{LB}}$, and initialize focal list $\mathcal{F} \leftarrow \{R\}$

  5: **while** $\mathcal{N}$ is not empty **do**

  6:      $N \leftarrow$ a CT node with the minimum d-value in $\mathcal{F}$

  7:      **if** $N_{\mathsf{Conf}} = \emptyset$ **then**

  8:          **return** N_{Sol}

  9:      Delete N from the open and focal lists

10:      Pick a conflict in N_{Conf}

11:      Generate 2 child CT nodes N^1 and N^2 of N, and update N^1_{Con} and N^2_{Con}

12:      Call low-level search for N^i to calculate N^i_{Sol}, N^i_{Cost} and N^i_{Conf} for $i = 1, 2$

13:      Add N^i to $\mathcal{N}$ if N^i_{Sol} exists for $i = 1, 2$

14:      Add N^i to $\mathcal{F}$ if N^i_{Sol} exists and $N^i_{\mathsf{Cost}} \leq w\mathsf{LB}$ for $i = 1, 2$

15:      **if** $min_{N \in \mathcal{N}} N_{\mathsf{LB}} > \mathsf{LB}$ **then**

16:          $\mathsf{LB} \leftarrow min_{N \in \mathcal{N}} N_{\mathsf{LB}}$

17:          $\mathcal{F} \leftarrow \{N \in \mathcal{N} : N_{\mathsf{LB}} \leq w\mathsf{LB}\}$

18: **return** No solution
---

path for a_i that respects the set of constraints of CT node N and is computed by the focal search. Let $N_{\mathsf{LB}} = \sum_{i=1}^{k} N_{\mathsf{LB},i}$. On the high level, ECBS performs a focal search (Lines 5 - 17) with a focal list that contains all CT nodes N in $\mathcal{N}$ such that $N_{\mathsf{Cost}} \leq w\mathsf{LB}$ (Lines 4, 14 and 17), where $\mathcal{N}$ is the open list and $\mathsf{LB} = \min_{N \in \mathcal{N}} N_{\mathsf{LB}}$ (Lines 4 and 16). Since LB is a lower bound on the sum of costs of any solution, once a solution is found by always expanding a CT node in the focal list, it is guaranteed to be a w-approximate solution. Selecting CT nodes from the focal list for expansions (Line 6) in the high level search of ECBS is an important decision to make. The common practice is to select a CT node with the minimum d-*value*, where the d-value is typically a hand-crafted value that is computed for each CT node when it is generated. The d-value of a CT node is an estimate of the effort required to find a solution in the CT subtree rooted at that CT node.

Algorithm 2 MAPF-LNS

1: **Input:** A MAPF instance I
2: $P = \{p_i : i \in [k]\} \leftarrow \mathsf{runInitialSolver}(I)$
3: Initialize the weights ω of the agent-set selection strategies
4: **while** runtime limit not exceeded **do**
5: $\quad \mathcal{H} \leftarrow \mathsf{selectDestroyHeuristic}(w)$
6: $\quad B \leftarrow \mathsf{selectAgentSet}(I, \mathcal{H})$
7: $\quad P^- \leftarrow \{p_i \in P : a_i \in B\}$
8: $\quad P^+ \leftarrow \mathsf{runReplanSolver}(I, B, P \setminus P^-)$
9: $\quad$ Update the weights ω of the agent-set selection strategies
10: $\quad$ **if** $\sum_{p \in P^+} l(p) < \sum_{p \in P^-} l(p)$ **then**
11: $\quad\quad P \leftarrow (P \setminus P^-) \cup P^+$
12: **return** P

2.3.3 MAPF-LNS

MAPF-LNS [117] is the state-of-the-art anytime MAPF search algorithm. It is able to solve large MAPF instances that most existing MAPF search algorithms fail to either solve or provide high-quality solutions for.

MAPF-LNS, shown in Algorithm 2, takes a MAPF instance as input and calls an efficient initial search algorithm to compute a solution P (Line 2). In each iteration, it selects an agent set B using an agent-set selection strategy $\mathcal{H}$ (Lines 5-6), deletes the paths P^- of the agents in B from P (Line 7) and calls a replan search algorithm to replan new paths P^+ for them that conflict with neither each other nor the paths in $P \setminus P^-$ (Line 8). If P^+ decreases the sum of costs (Line 10), then MAPF-LNS replaces P^- with P^+ (Line 11). The initial search algorithm could be any off-the-shelf MAPF search algorithm, and the replan search algorithm could be any off-the-shelf MAPF search algorithm that can handle moving obstacles.

MAPF-LNS uses two randomized agent-set selection strategies, namely an agent-based heuristic and a map-based heuristic, to generate the agent sets B. The agent-based heuristic generates the agent set B by including the agent a_i with the largest delay and other agents (found via a random walk procedure) whose paths prevent it from achieving a lower agent cost. The map-based heuristic randomly chooses a vertex with a degree greater than 2 in graph G and generates the agent set B by including some of the agents whose paths visit the chosen vertex. Both the agent-based and the map-based heuristics impose a limit

on the cardinality of the agent set. MAPF-LNS uses Adaptive LNS [166], essentially an online learning algorithm, to select one of the two agent-set selection strategies by maintaining a weight for each of them.

2.3.4 Prioritized Planning (PP)

Definition 2.3.1 (Priority ordering). *A priority ordering $\prec$ is a strict partial order on $\{a_1, \ldots, a_k\}$: $a_i \prec a_j$ iff agent a_i has higher priority than agent a_j [137]. $\prec$ is a* total *priority ordering iff any two agents in $\{a_1, \ldots, a_k\}$ are comparable (i.e., either $a_i \prec a_j$ or $a_j \prec a_i$ for all $a_i \neq a_j$) and a* partial *priority ordering otherwise.*

Prioritized Planning (PP) [49] is an unbounded-suboptimal MAPF search algorithm. In PP, we arrange all agents into a predefined total priority ordering. Then, we plan paths for all agents one by one in descending order according to the priority ordering. The path of each agent is the individually cost-minimal path from its start vertex to its goal vertex that has no conflicts with the paths of all higher-priority agents. So, instead of planning paths for all agents at once, PP decouples the planning process and plans for the agents sequentially. PP does not guarantee completeness or optimality, but it is popular because of its efficiency and simplicity. A key consideration in PP is how to determine the predefined total priority ordering. It is typically determined either randomly or via manually designed heuristics. We introduce a few of them in the following.

Query-Distance Heuristic [17] proposes the query-distance heuristic, which measures the start-goal graph distance $dist(s_i, g_i)$ of each agent a_i and assigns higher priority to agents with longer distances. The motivation behind this heuristic was to prioritize agents that need to travel longer distances and thus minimize the makespan (i.e., the largest cost of all agents). An opposite version of the query-distance heuristic, which assigns higher priority to agents with shorter start-goal graph distances, has been used in [137].

Least-Option Heuristic Building on the idea behind the most-constrained-variable heuristic for solving constraint satisfaction problems, [191] and [196] propose the least-option heuristic, which assigns higher priority to agents with fewer path options, where the number of path options for an agent is defined as the number of paths that do not have conflicts with the paths of already-planned agents within a given number of time steps in [191] or the number of homology classes of paths in [196].

Start-and-Goal-Conflict Heuristic [24] propose prioritization rules that consider the potential conflicts at the start and goal vertices of the agents. Intuitively, if the individually cost-minimal path of agent a_i visits the start vertex of another agent a_j, then a_j needs to be planned prior to a_i; if the individually cost-minimal path of agent a_i visits the goal vertex of another agent a_j, then a_i needs to be planned prior to a_j. This heuristic tends to reduce the runtime of PP [24] and increase its success rate [18].

Random Restarts When the priority ordering is assigned randomly, people often apply random restarts to improve the performance of PP [16]. When PP with a particular priority ordering fails to find a solution for a MAPF instance, we can "restart" it with a new randomized priority ordering.

2.3.5 MAPF Instances Used in the Empirical Evaluation

In our empirical evaluation, we use grid maps from the following sources: (1) grid maps from the MAPF benchmark [181]; (2) grid map "'lak503d'" from the 2D pathfinding benchmarks [182]; and (3) random maps and warehouse maps that we generate following [120], where random maps are grid maps with randomly blocked cells and warehouse maps are grid maps with rectangular obstacles (clusters of blocked cells).

We describe the training instances, validation instances and test instances used in our empirical evaluation. For each grid map from the MAPF benchmark [181], we use the 25 random scenarios provided. A scenario is a list of randomly created pairs of start and goal vertices for a given grid map. Given a grid

map M and a number of agents k, we generate 25 test instances $\mathcal{I}_{\text{Test}}^{(M)}$, one from each scenario, by using

the first k pairs of start and goal vertices. In order to generate training instances and validation instances

that follow a similar distribution as the test instances, given a scenario with a map $M \in \mathcal{M}$ and a number

of agents k, we generate a training instance $I \in \mathcal{I}_{\text{Train}}^{(M)}$ by randomly selecting k start vertices from all start

vertices in the scenario, randomly selecting k goal vertices from all goal vertices in the scenario and then

randomly combining them into k pairs of start and goal vertices. For grid maps that are not from the MAPF

benchmark, we generate the MAPF instances in the same way.

The numbers of agents and the types of grid maps used in the MAPF instances and the runtime limits in

our empirical evaluation are mainly based on the scalability of the MAPF search algorithms. For example,

CBS is an optimal MAPF search algorithm, which is slow and does not scale compared to ECBS, MAPF-

LNS or PP. Thus, we use MAPF instances on smaller grid maps with lower agent and/or obstacle densities

and use longer runtime limits. In contrast, for MAPF-LNS and PP, we use MAPF instances on larger maps

with higher agent and/or obstacle densities (that is, easier MAPF instances) and use shorter runtime limits.

These MAPF instance configurations are also impacted by the amount of computation resources we had

access to, which changed from time to time. When we had limited computation resources available, we

tended to use shorter runtime limits and easier MAPF instances.

All empirical evaluations in this chapter follow the setup described above unless stated otherwise.

2.4   Related Work

In this section, we summarize related works on MAPF search algorithms and using ML for MAPF. Finally,

we discuss ML for other combinatorial optimization problems (COP) that inspires our work.

2.4.1 MAPF Search Algorithms

For optimal MAPF search algorithms, there has been a huge effort to improve CBS. Selecting which conflict to resolve next has been explored in Improved CBS (ICBS) [22]. ICBS categorizes conflicts into three types to prioritize them: (i) A conflict is cardinal iff, when CBS uses the conflict to split CT node N, the costs of both resulting child nodes are strictly larger than N_{Cost}; (ii) a conflict is semi-cardinal iff the cost of one of the child nodes is strictly larger than N_{Cost} and the cost of the other child node is the same as N_{Cost}; and (iii) a conflict is non-cardinal otherwise. By first resolving cardinal conflicts, then semi-cardinal conflicts and finally non-cardinal conflicts, CBS is able to improve its efficiency since it increases the lower bound on the optimal cost more quickly by generating child nodes with larger costs. ICBS uses Multi-Valued Decision Diagrams (MDDs) to classify conflicts. An MDD for agent a_i is a directed acyclic graph consisting of all cost-minimal paths from s_i to t_i that respect the current constraints N_{Con}. The vertices at level t of the MDD are exactly the vertices that agent a_i could be at when following one of its cost-minimal paths. The width of an MDD level is the number of vertices at that level and a singleton is an MDD level with width one. A vertex conflict $\langle a_i, a_j, v, t \rangle$ (edge conflict $\langle a_i, a_j, u, v, t \rangle$) is cardinal iff vertex v (edge (u, v)) is the only vertex at depth t (the only edge from depth t to depth $t + 1$) in the MDDs of both agents. [122] proposes to split the search space into two disjoint ones when expanding a CT node in CBS and prioritizes conflicts based on the number of singletons in or the widths of level t of the MDDs of both agents. To the best of our knowledge, other than selecting conflicts using MDDs, conflict selection for CBS has not yet been explored. In this chapter, we show how one can apply ML to learn an improved conflict-selection strategy in CBSH2 [120] for CBS-based optimal MAPF search algorithms. CBSH2 is the state-of-the-art version of CBS and uses the same conflict-selection strategy as ICBS.

Another line of research focuses on speeding up CBS by calculating a tighter lower bound on the optimal cost to guide the high-level search. When expanding a CT node N, CBSH [54] uses the CG heuristic, which builds a conflict graph (CG) whose vertices represent agents and whose edges represent cardinal

conflicts in N_Sol. Then, the lower bound on the optimal cost within the subtree rooted at N is guaranteed to increase at least by the size of the minimum vertex cover of this CG. We refer to this increment as the *h-value* of the CT node. Based on CBSH, CBSH2 [120] uses the DG and WDG heuristics that generalize CG and compute h-values for CT nodes using (weighted) pairwise dependency graphs that take into account semi-cardinal and non-cardinal conflicts besides cardinal ones. CBSH2 with the WDG heuristic is the current state-of-the-art CBS-based optimal MAPF search algorithm [120]. CBSH2 uses the same conflict-selection strategy proposed in Improved CBS [22].

There are optimal compilation-based MAPF search algorithms that reduce MAPF to other COPs. Branch-and-cut-and-price [112, 111] is a bi-level MAPF search algorithm based on MILP solvers. MAPF has also been encoded as satisfiability [183], constraint programming [57] and answer set programming problems [63].

For bounded-suboptimal MAPF search algorithms, there are A*-based algorithms, such as Enhanced Partial-Expansion A* [53], A* with operator decomposition [180] and M* [190], and CBS-based algorithms, such as ECBS [10] and EECBS [124]. Variants of ECBS, such as ECBS with the highway heuristic [35] and Improved ECBS [36], have been proposed to speed up ECBS by generating highways (paths for the agents that include edges from user-provided sets of edges) in environments such as warehouses. However, these approaches are not generalizable to environments with open areas or environments without straight corridors. EECBS [124] is the current state-of-the-art bounded-suboptimal MAPF search algorithm. It replaces focal search in ECBS with explicit estimation search [186], that selects the next node to expand the CT from three different lists based on certain rules. In contrast, ECBS selects nodes only from the focal list, which is a simpler rule. For demonstration purposes, in this chapter, we use ECBS as an example to show how one can apply ML to learn an improved node-selection strategy for CBS-based bounded-suboptimal MAPF search algorithms.

For unbounded-suboptimal MAPF search algorithms, there are prioritized planning-based algorithms, such as prioritized planning, and rule-based algorithms, such as Push-and-Swap [135], PPS [170] and Priority Inheritance with Backtracking [150]. MAPF-LNS [117] and MAPF-LNS2 [119] are the state-of-the-art unbounded-suboptimal MAPF search algorithms based on LNS. MAPF-LNS starts with finding an initial solution fast and then iteratively improves it over time. MAPF-LNS2 starts with a set of paths with conflicts, then iteratively reduces the number of conflicts to find a solution and finally uses MAPF-LNS to optimize this solution. For demonstration purposes, in this chapter, we use MAPF-LNS to show how one can apply ML to learn an improved agent-set selection strategy for LNS-based unbounded-suboptimal MAPF search algorithms. Although the solution qualities of rule-based and prioritized planning-based algorithms are often worse than those of other types of MAPF search algorithms, they run in polynomial time. Therefore, they can compute a solution quickly and are quite popular. In this chapter, we also show how one can apply ML to learn a priority-assignment strategy for PP.

2.4.2   ML for MAPF

Our work is one of the first to use ML to improve decision-making within MAPF search algorithms. ML has been applied to decentralized MAPF, where agents coordinate in a decentralized fashion. [172] proposes a framework that combines reinforcement learning and imitation learning to learn decentralized policies for agents to avoid expensive centralized planning. An enhanced version [38] is later proposed that resolves deadlocks in congested environments using symmetry-breaking techniques. [141, 128, 193] show that the communication capabilities of agents help further resolve deadlocks and congestion. ML has also been used to select the best MAPF search algorithms for solving MAPF optimally [98, 163] and suboptimally [31].

2.4.3 ML for other COPs that Inspires Our Work

Using ML to improve combinatorial search has been studied extensively for other COPs. Mixed integer linear programs (MILP) are powerful tools for modeling and solving a wide variety of COPs. There is a huge body of studies that use ML to improve decision-making in Branch-and-Bound (BnB) search [194] for MILPs, and our works have been inspired by some of them. BnB is a tree search, and, as part of BnB, nodes in the search tree that contain unassigned variables must be expanded into two child nodes by selecting one of the unassigned variables and splitting its domain by adding new constraints. There has been a line of works on learning how to select variables to branch on for BnB [102, 59, 68, 211], where the main goal is commonly to imitate the effective but expensive Strong Branching heuristic [7]. Conflict selection in CBS is similar to variable selection in BnB, but previous methods are not directly applicable since the Strong Branching heuristic does not apply to CBS and features used in those works are based on variables and constraints that are specific to MILPs. We thus leverage insights from MAPF to craft our methods. In addition, learning to select nodes to expand [73, 178, 110] in BnB and learning to select variables to reoptimize in LNS [177] have been explored for solving MILPs. For node selection in BnB, [73] uses imitation learning to learn node-selection and node-pruning strategies for solving MILPs. [178] scale up this approach by progressively increasing the instance sizes in the form of curriculum learning. For variable selection in LNS, both [177] and [179] use imitation learning where [177] learns to imitate an expert based on random sampling and [179] learns to imitate the Local Branching heuristic [56], which is a more effective one. Inspired by these early works, we seek to improve similar decisions for ECBS and MAPF-LNS and develop methods that work for our specific tasks in the context of MAPF. We show how we achieve those tasks by leveraging domain expertise and designing engineering techniques that suit MAPF search algorithms.

2.5 An Imitation Learning Framework for Learning Decision-Making Strategies

In this section, we propose a general ML framework based on imitation learning to learn decision-making strategies for MAPF search algorithms. As will be shown in Sections 2.6-2.9, based on our framework, we develop the first ML methods to improve centralized MAPF search algorithms. It is also the first ML framework in the literature that has been successfully applied to improving multiple MAPF search algorithms. It consists of the following steps:

1. **Identify a Decision to Improve** Given a MAPF search algorithm, identify a decision that is crucial to its performance. The goal is to learn a strategy to improve making this decision.

2. **Find an Expert** Since our framework is based on imitation learning, this step identifies an expert to provide high-quality demonstrations for decision-making so that we can imitate it via supervised learning. Formally, we define a state as a snapshot of the search whenever a decision needs to be made and the set of actions $A(s)$ at state s as the set of possible decisions at s. An expert evaluates all actions $A(s)$ and selects the optimal or suboptimal action $a^* \in A(s)$ as the decision. Compared to a trivial strategy (for example, a greedy strategy or a random choice strategy), an ideal expert makes more effective decisions at a slightly higher but reasonable computational cost. The computational cost should not be too low, otherwise one could potentially simply deploy the expert in the search to get good solutions with low runtime. It should not be too high either since the expert will be used to collect demonstrations as labels for supervised learning and we need a sufficient amount of them to train an ML model. Since the expert evaluates multiple actions, it provides not only the best decisions but also information on low-quality decisions.

3. **Data Collection** We obtain a training dataset D, which is a set of states. At each state s of the search, we can compute features and labels for each available action $a \in A(s)$. By observing and

recording the features and the decisions given by the expert, we then learn to make predictions as similar as possible to the expert without actually probing it. Features serve as signals to inform predictions and should be fast to compute. The features depend on the choice of ML model. For example, image-like features are more suitable for convolutional neural networks, and graph representations are needed for graph neural networks. Lightweight models, such as linear regression models, support vector machines (SVM) and decision trees, are sometimes more favorable for repeated decision-making since they have much lower computational overhead. For each decision, labels serve as the learning target and are derived from the quality of actions determined by the expert. Given the expert, one could label the actions as good or bad actions based on a performance metric, such as the solution quality or runtime they lead to, or a proxy for the metric.

4. **Model Learning** We reduce the ML problem to an imitation learning task, that is, to imitate the expert. There are a few methods for imitation learning. One could learn to classify actions based on their labels or predict the labels. In our work, we deploy a learning-to-rank method, where we learn to rank the actions based on their rankings derived from the labels. The main benefit of learning-to-rank is that it learns to predict the actions of the expert from the differences among actions, which has been shown effective [73, 102].

5. **ML-Guided Search** Once we have a trained ML model, we plug it into the MAPF search algorithm as a decision-making strategy.

In the rest of this section, we formulate the imitation-learning task in Step 4 as a learning-to-rank task. For a state s and an action $a \in A(s)$, let $\phi_s(a) \in \mathbb{R}^p$ be the feature vector of action $a \in A(s)$ and $y_s(a)$ be the ground truth label for a. The goal is to learn a ranking function $\pi : \mathbb{R}^p \to \mathbb{R}$ that serves as a scoring function for each action, where better actions receive higher scores. π takes as input the p-dimensional features $\phi_s(a)$ of action a at state s and then predicts $a^* = \arg\max_{a \in A(s)} \hat{y}_s(a)$ as the best action, where

$\hat{y}_s(a) = \pi(\phi_s(a))$ is the predicted score. π could be learned by regression on the labels or classifying the labels. However, such methods learn to score each action independently. In contrast to a regression or classification task, we adopt a formulation of learning-to-rank that predicts the score based on the relative differences between pairs of actions instead of based on the action itself. The labels must be numerical values that indicate the qualities of the actions, such as runtimes or solution qualities. The loss function for training is based on a strict partial order on $A(s)$ derived from the labels. Formally, given a set of states as the training data D, we minimize the following loss function

$$L(\boldsymbol{w}) = \sum_{s \in D} l(y_s, \hat{y}_s) + \frac{C}{2} ||\boldsymbol{w}||_2^2,$$

where $\boldsymbol{w}$ are the learnable weights of π, C is a regularization parameter and $l(y_s, \hat{y}_s)$ is a loss function based on a pairwise loss between the ground truth labels y_s and the predicted scores $\hat{y}_s$. To compute $l(y_s, \hat{y}_s)$, we consider the set of ordered pairs of actions at state s where one of their labels is greater than the other

$$\mathcal{P}_s = \{(a', a'') : y_s(a') > y_s(a'') \wedge a', a'' \in A(s)\}.$$

$l(y_s, \hat{y}_s)$ is the weighted fraction of swapped pairs in the predictions defined as

$$l(y_s, \hat{y}_s) = \frac{\sum_{(a', a'') \in \mathcal{P}_s : \hat{y}_s(a') \leq \hat{y}_s(a'')} \tilde{w}_{a', a''}}{\sum_{(a', a'') \in \mathcal{P}_s} \tilde{w}_{a', a''}},$$

where $\tilde{w}_{a', a''}$ is a weight associated with each pair of actions in $\mathcal{P}_s$. One could simply set $\tilde{w}_{a', a''}$ to a constant for an unweighted version of $l(y_s, \hat{y}_s)$.

To learn π, one could train a deep neural network (DNN), like most of the existing works in the literature, such as RankNet [25] and LambdaRank [162]. However, DNNs introduce an undesirably large

computational overhead if used for repeated decision-making in the MAPF search algorithms. We thus learn a linear ranking function

$$\pi : \mathbb{R}^p \to \mathbb{R} : \pi(\phi_s(a)) = \boldsymbol{w}^T \phi_s(a)$$

using SVM$^{\mathsf{rank}}$ [94] instead, which has been shown to be efficient and effective in previous work on learning to rank [102, 73] in the context of search algorithms for other COPs. In practice, one can train the linear ranking function with open-source solvers such as those developed for SVM$^{\mathsf{rank}}$ [95] and LIBLINEAR [52]. These solvers minimize an upper bound on the loss, since the loss itself is NP-hard to minimize. In our implementation, we use SVM$^{\mathsf{rank}}$ [94] for the unweighted version of $l(y_s, \hat{y}_s)$ and use LIBLINEAR [52] otherwise, since SVM$^{\mathsf{rank}}$ is simpler to use but does not allow customizing the weights.

2.6 Learning to Select Conflicts for CBS

In this section, we introduce CBS+ML to show how the framework can be applied to improve conflict selection in CBS [174]. CBS is one of the leading algorithms for solving MAPF optimally, and a number of enhancements to CBS have been developed [22, 120, 54, 10]. Picking good conflicts is important, and a good strategy for conflict selection could have a big impact on the efficiency of CBS by reducing both the size of the CT and its runtime. We refer the readers to the example in Figure 1 in [22] that demonstrates why the size of the CT can be impacted by conflict selections. To the best of our knowledge, other than prioritizing conflicts using MDDs in ICBS [22], conflict prioritization has not yet been explored much. In the rest of this section, we apply the framework introduced in Section 2.5 and propose CBS+ML tailored to conflict selection in CBS. We then empirically demonstrate the effectiveness and efficiency of CBS+ML.

2.6.1 Machine Learning Methodololy

Our goal is to learn a conflict-selection strategy that reduces the runtime of CBS. The conflict-selection strategy is applied when expanding a CT node N. Thus, we represent the state of the search with the CT node $s = N$. The state contains information about not only CT node N but also its ancestors and siblings that are generated before expanding N. CBS can select any conflict from N_{Conf} to resolve. Thus, the available actions at CT node N are $A(N) = N_{\mathsf{Conf}}$. To apply the framework, we first propose an expert for conflict selection that results in smaller CT sizes than the ones used in previous work. However, the expert is much more computationally expensive since it has to compute the WDG heuristic for each conflict that will be explained in Section 2.6.1.1. Next, given the expert, we explain how we use ML to imitate its decisions.

2.6.1.1 Experts for Conflict Selection

Given a MAPF instance, an expert for conflict selection at a particular CT node N is a ranking function that takes the set of conflicts N_{Conf} as input, calculates a real-valued score for each conflict and outputs the ranks determined by the scores. We say that CBS follows an expert for conflict selection iff CBS builds the CT by always resolving the conflict with the highest rank. We define expert O_0 as the one proposed in ICBS [22], which uses MDDs to rank conflicts.

Definition 2.6.1. *Given a CT node N, expert O_0 ranks the conflicts in N_{Conf} in the order of cardinal conflicts, semi-cardinal conflicts and non-cardinal conflicts, breaking ties in favor of conflicts at the smallest time step and remaining ties randomly.*

Next, we define experts O_1 and O_2, that both calculate 1-step lookahead scores by using, for each conflict, the two child nodes of N that would result if the conflict were resolved at N.

Definition 2.6.2. *Given a CT node N, expert O_1 computes the score $v_c = \min\{g_c^l + h_c^l, g_c^r + h_c^r\}$ for each conflict $c \in N_{\mathsf{Conf}}$, where g_c^l and g_c^r would be the costs of the two child nodes of N and h_c^l and h_c^r would be*

	The Random Map			
	Runtime	CT Size	Expert Time	Search Time
CBSH2+O_0	9.95s	2,362 nodes	**0.00s**	9.95s
CBSH2+O_1	24.89s	746 nodes	21.34s	3.55s
CBSH2+O_2	12.13s	**632 nodes**	9.52s	**2.61s**
CBS+ML	**6.19s**	998 nodes	0.88s	5.31s
	The Game Map			
	Runtime	CT Size	Expert Time	Search Time
CBSH2+O_0	2.3min	952 nodes	**0.0min**	2.3min
CBSH2+O_1	19.8min	**565 nodes**	19.0min	**0.8min**
CBSH2+O_2	27.4min	2,252 nodes	23.4min	4.0min
CBS+ML	**1.6min**	754 nodes	0.2min	1.4min

Table 2.1: Performance of CBSH2 with different experts and our method CBS+ML. Expert time is the runtime of the expert. Search time is the runtime minus the expert time. All entries are averaged over the MAPF instances that are solved by all methods.

the h-values given by the WDG heuristic for the two child nodes of N if conflict c were resolved at N. Then, it

outputs the ranks determined by the decreasing order of the scores (i.e., the highest rank for the highest score).

Expert O_1 selects the conflict that results in the tightest lower bound on the optimal cost estimated by the WDG heuristic in the child nodes. Inspired by CBSH2, we use the WDG heuristic in expert O_1 to compute the h-values since it is the state of the art. The intuition behind using this expert is that the sum of the cost and the h-value of a node is a lower bound on the cost of any solution found in the subtree rooted in the node, and, sometimes, the lower bound maintained by CBS might not increase for most conflicts you select. Thus, we want CBS to increase the lower bound as much as possible to find a solution quickly by selecting the right conflict to resolve next.

Definition 2.6.3. *Given a CT node N, expert O_2 computes the score $v_c = \min\{m_c^l, m_c^r\}$ for each conflict $c \in N_{\mathsf{Conf}}$, where m_c^l and m_c^r would be the number of remaining conflicts in the two child nodes of N if conflict c were resolved at N. Then, it outputs the ranks determined by the increasing order of the scores (i.e., the highest rank for the lowest score).*

Expert O_2 selects the conflict that results in the least number of conflicts in the child nodes.

We use CBSH2 with the WDG heuristic as our search algorithm and run it with experts O_0, O_1 and O_2 on (1) the random map, which is a 20×20 four-neighbor grid map with 25% randomly generated blocked cells [120], and (2) the game map "lak503d" [182], which is a 192×192 four-neighbor grid map with 51% blocked cells from the video game *Dragon Age: Origins*. The maps are shown in Table 2.4. The experiments are conducted on 2.4 GHz Intel Core i7 CPUs with 16 GB RAM. We set the runtime limit to 20 minutes for the random map and 1 hour for the game map. We set the number of agents to $k = 18$ for the random map and $k = 100$ for the game map and run each variant of CBSH2 on 50 MAPF instances for each grid map. The MAPF instances are generated the same way for training instances as in Section 2.3.5. In Table 2.1, we present the performance of the three experts as well as our method CBS+ML. All entries are averaged over the MAPF instances that are solved by all methods. We evaluate the experts according to the resulting CT sizes since they determine the runtime when the calculation of the experts is not taken into account (and everything else being equal) and first look at the CT sizes of CBSH2 with each of the three experts. Expert O_2 is best for the random map, followed closely by expert O_1. Expert O_1 is best for the game map. Overall, expert O_1 is best. Therefore, in the rest of the chapter, we mainly focus on learning a ranking function that imitates expert O_1. Table 2.1 shows that, by learning to imitate expert O_1, our method CBS+ML[†] achieves the best runtime, even though it induces a larger CT than CBSH2+O_1. Next, we introduce our ML methodology.

2.6.1.2 Data Collection

The next step in our framework is to construct a training dataset from which we can learn a model that imitates the expert's output. First, we fix the graph underlying the MAPF instances that we want to solve and the number of agents. The number of agents is only fixed during the data collection and model learning steps. Later in Section 2.6.2, we show that the models can generalize to MAPF instances with larger numbers of agents during testing. We obtain a set of MAPF instances $\mathcal{I}_{\text{Train}}$ for training. A MAPF instance

---

[†]This is the CBS+ML-S variant that will be introduced in Section 2.6.2.

dataset D_I is obtained for each $I \in \mathcal{I}_{\mathsf{Train}}$, and the final training dataset is obtained by taking the union of these datasets $D = \bigcup_{I \in \mathcal{I}_{\mathsf{Train}}} D_I$. To obtain dataset D_I, we run CBSH2 on I and expert O_1 is run for each CT node N to produce the ranking for N_{Conf}. D_I consists of a set of those CT nodes $\mathcal{N}$ which are the expanded CT nodes during the search. For each node $N \in D_I$, the conflicts in N_{Conf} are the available actions $A(N)$. For each $c \in N_{\mathsf{Conf}}$, we compute a binary label $y_N(c) \in \{0, 1\}$ derived from the expert's ranking of the conflicts and a p-dimensional feature vector $\phi_N(c)$ that describes the characteristics of the conflict c at CT node N. We also collect a validation dataset on another set of MAPF instances $\mathcal{I}_{\mathsf{Valid}}$ for validation to evaluate the prediction accuracy of the learned model.

Features   We collect a p-dimensional feature vector $\phi_N(c)$ that describes a conflict $c \in N_{\mathsf{Conf}}$ in CT node N. The $p = 67$ features of a conflict $\langle a_i, a_j, v, t \rangle$ ($\langle a_i, a_j, u, v, t \rangle$) in our implementation are summarized in Table 2.2. They consist of (1) the properties of the conflict, (2) statistics of CT node N, the conflicting agents a_i and a_j and the contested vertex or edge with respect to N_{Sol}, (3) the number of conflicts that have been resolved for a vertex[‡] or an agent, and (4) features of the MDD and the WDG. We perform a linear transformation to normalize the value of each feature to the range of $[0, 1]$ across all conflicts in N_{Conf}, where the minimum value of that feature gets transformed into a 0 and the maximum value gets transformed into a 1. All features of a given conflict $c \in N_{\mathsf{Conf}}$ can be computed in $O(|N_{\mathsf{Conf}}| + k)$ time.

Labels   We label each conflict in N_{Conf} such that conflicts with higher ranks determined by the expert have larger labels. Instead of using the full ranking provided by expert O_1, we use a binary labeling scheme similar to the one proposed by [102]. We assign label 1 to a conflict if no more than 20% of the conflicts in N_{Conf} have the same or a higher score; otherwise, we assign label 0 to it. When more than 20% of the conflicts have the same highest O_1 score, we assign label 1 to those conflicts and label 0 to the rest. By doing so, we ensure that at least one conflict is labeled 1 and conflicts with the same score have the

---

[‡]An edge conflict is considered to be resolved for both vertices on the edge.

Feature Descriptions	Count
Types of the conflict: binary indicators for edge conflicts, vertex conflicts, cardinal conflicts, semi-cardinal conflicts and non-cardinal conflicts.	5
Number of conflicts involving agent a_i (a_j) that have been selected and resolved so far during the search: their minimum, maximum and sum.	3
Number of conflicts that have been selected and resolved so far during the search at vertex u (v): their minimum, maximum and sum.	3
Number of conflicts that agent a_i (a_j) is involved in: their minimum, maximum and sum.	3
Time step t of the conflict.	1
Ratio of t and the makespan of N_{Sol}.	1
Cost of the path of agent a_i (a_j) in N_{Sol}: their minimum, maximum, sum, absolute difference and ratio of their maximum and minimum.	5
Delay of agent a_i (a_j): their minimum and maximum.	2
Ratio of the costs of the path of agent a_i (a_j) and its individually cost-minimal path: their minimum and maximum.	2
Difference of the cost of the path of agent a_i (a_j) and t: their minimum and maximum.	2
Ratio of the cost of the path of agent a_i (a_j) and t: their minimum and maximum.	2
Ratio of the cost of the path of agent a_i (a_j) and N_{Cost}: their minimum and maximum.	2
Binary indicator whether none (at least one) of agents a_i and a_j has reached its goal vertex by time step t.	2
Number of conflicts $c' \in N_{\mathsf{Conf}}$ such that $\min\{d_{q,q'} : q \in V_c^T, q' \in V_{c'}^T\} = w$ ($0 \le w \le 5$).	6
Number of agents a such that there exists $q' \in V_a$ and $q \in V_c^T$ such that $d_{q,q'} = w$ ($0 \le w \le 5$).	6
Number of conflicts $c' \in N_{\mathsf{Conf}}$ such that $\min\{d_{q,q'} : q \in V_c, q' \in V_{c'}\} = w$ ($0 \le w \le 5$).	6
Width of level w ($|w - t| \le 2$) of the MDD for agent $a_i(a_j)$ (we use zero as the width of a level that does not exist): their minimum and maximum [122].	10
Weight of the edge between agents a_i and a_j in the weighted dependency graph [120].	1
Number of vertices q' in graph G such that $\min\{d_{q',q} : q \in V_c\} = w$ ($1 \le w \le 5$).	5

Table 2.2: Features of a conflict $c = \langle a_i, a_j, u, t \rangle$ ($\langle a_i, a_j, u, v, t \rangle$) of a CT node N. Given the underlying graph $G = (V, E)$, let $V_T = \{(v, t) : v \in V, t \in \mathbb{Z}_{\ge 0}\}$, $E_T = \{((u, t), (v, t + 1)) : t \in \mathbb{Z}_{\ge 0} \wedge (u = v \vee (u, v) \in E)\}$, and define the time-expanded graph as an unweighted graph $G_T = (V_T, E_T)$. Let $d_{u,v}$ be the cost of the cost-minimal path between vertices u and v in G and $d_{(u',t'),(u,t)}$ be the distance from (u', t') to (u, t) in G_T if $t' \le t$ or from (u, t) to (u', t'), otherwise. For a conflict $c' = \langle a_i', a_j', u', t' \rangle$ ($\langle a_i', a_j', u', v', t' \rangle$) in N_{Conf}, define $V_{c'} = \{u'\}$ ($V_{c'} = \{u', v'\}$) and $V_{c'}^T = \{(u', t')\}$ ($V_{c'}^T = \{(u', t'), (v', t')\}$). For an agent a, define $V_a = \{(u, t) : \text{agent } a \text{ is at vertex } u \text{ at time step } t \text{ following its path}\}$. The counts are the numbers of features contributed by the corresponding entries, which add up to $p = 67$.

same label. This labeling scheme relaxes the definition of "top" conflicts that allows the learning algorithm to focus on only high-ranking conflicts and avoids the irrelevant task of learning the correct ranking of conflicts with low scores. We tried directly using their O_1 scores as their labels but did not get as good performance as using the scheme described.

2.6.1.3 Model Learning

Given the training dataset $D = \bigcup_{I \in \mathcal{I}_{\mathsf{Train}}} D_I$, we follow the formulation in Section 2.5 to learn a linear ranking function with parameter $\boldsymbol{w} \in \mathbb{R}^p$

$$\pi : \mathbb{R}^p \to \mathbb{R} : \pi(\phi_N(c)) = \boldsymbol{w}^{\mathsf{T}} \phi_N(c)$$

that minimizes the loss function

$$L(\boldsymbol{w}) = \sum_{N \in D} l(y_N, \hat{y}_N) + \frac{C}{2} ||\boldsymbol{w}||_2^2.$$

To compute $l(y_N, \hat{y}_N)$, we consider the set of pairs $\mathcal{P}_N = \{(c_i, c_j) : c_i, c_j \in N_{\mathsf{Conf}} \wedge y_N(c_i) > y_N(c_j))\}$, where $\hat{y}_N(c) = \pi(\phi_N(c))$ is the predicted scores for conflict c. The loss function $l(\cdot, \cdot)$ is the fraction of swapped pairs, computed as

$$l(y_N, \hat{y}_N) = \frac{1}{|\mathcal{P}_N|} |\{(c_i, c_j) \in \mathcal{P}_N : \hat{y}_N(c_i) \leq \hat{y}_N(c_j)\}|.$$

2.6.1.4 ML-Guided Search

After data collection and model learning, we replace expert O_1 for conflict selection in CBS with the learned ranking function $\pi(\cdot)$. At each CT node N, we first compute the feature vector $\phi_N(c)$ for each conflict $c \in N_{\mathsf{Conf}}$ and pick the conflict with the maximum score $c^* = \arg\max_{c \in N_{\mathsf{Conf}}} \pi(\phi_N(c))$. The time

complexity of conflict selection at node N is $O(|N_{\mathsf{Conf}}|(|N_{\mathsf{Conf}}| + k))^{\S}$. Even though the complexity of conflict selection with expert O_0 is only $O(|N_{\mathsf{Conf}}|)$, we will show in our experiments that we are able to outperform CBSH2+O_0 in terms of both the CT size and the runtime.

Discussion Improving conflict-selection strategies with ML in CBS is inspired by previous works [102, 59] on improving variable-selection strategies with ML in BnB for MILPs. We leverage several techniques and insights from previous works and tailor them for CBS. First, we leverage imitation learning to imitate decisions made by an expert. For MILP, [102, 59] imitate the Strong Branching [7] heuristic that solves a linear program relaxation for each candidate variable to estimate the increase on the lower bound in the resulting child nodes if it is branched on. For MAPF, one of the main contributions is that we design expert O_1 similarly to estimate the increase on the lower bound in the resulting child CT nodes for each conflict if it is selected. Second, we leverage a linear ranking function as our ML model. For MILP, both linear ML models [102] and deep neural networks [59] have been used. For MAPF, we choose to use the linear model since MAPF search algorithms are a lot more sensitive to the inference runtime (i.e., the runtime for computing the features and predictions) than MILP search algorithms. Therefore, we leverage domain expertise to craft features for the linear model that suit MAPF search algorithms. Third, we leverage the labeling scheme that relaxes the definition of "top" conflicts. Such a scheme is more effective than labeling each conflict with its score given by the expert, which aligns with the observation in MILP solving by [102].

2.6.2 Empirical Evaluation

In this subsection, we demonstrate the efficiency of CBS+ML through experiments. In the following, we introduce our evaluation setup and then present the results.

---

§We exclude the time complexity of building the MDDs for both CBS+ML and CBSH2+O_0.

2.6.2.1 Setup

We use the C++ code for CBSH2 with the WDG heuristic [120] as our CBS version. We compare against CBSH2+O_0 as a baseline since O_0 is the most commonly used conflict-selection expert. The reason why we choose CBSH2 with the WDG heuristic over CBS, ICBS and CBSH2 with the CG or DG heuristics is that it performs best, as demonstrated in [120]. We use the same compute resources as described in Section 2.6.1.1.

Our experiments provide answers to the following questions:

1. If the graph underlying the MAPF instances is known in advance, can we learn a model that performs well on unseen MAPF instances on the same graph with different numbers of agents?

2. If the graph underlying the MAPF instances is unknown in advance, can we learn a model from other graphs that performs well on MAPF instances on that graph?

We use a set of five four-neighbor grid maps $\mathcal{M}$ of different sizes and structures as the graphs underlying the MAPF instances and evaluate our algorithms on them. $\mathcal{M}$ includes (1) a warehouse map [121], which is a 79×31 grid map with 100 6×2 rectangular obstacles; (2) the room map "room-32-32-4" [181], which is a 32×32 grid map with 64 3×3 rooms connected by single-cell doors; (3) the random map; (4) the city map "Paris_1_256" [181], which is a 256×256 grid map of Paris; (5) the game map. The figures of the maps are shown in Table 2.4. For each grid map $M \in \mathcal{M}$, we collect data for training instances $\mathcal{I}_{\mathsf{Train}}^{(M)}$ and validation instances $\mathcal{I}_{\mathsf{Valid}}^{(M)}$ on M with a fixed number of agents, where $|\mathcal{I}_{\mathsf{Train}}^{(M)}| = 30$ and $|\mathcal{I}_{\mathsf{Valid}}^{(M)}| = 20$. We learn two ranking functions for grid map M: one ranking function that is trained using 5,000 CT nodes i.i.d. sampled from the training dataset collected by solving training instances $\mathcal{I}_{\mathsf{Train}}^{(M)}$ on the same grid map and another one that is trained using 5,000 CT nodes sampled from the training dataset collected by solving training instances $\cup_{M' \in \mathcal{M}} \mathcal{I}_{\mathsf{Train}}^{(M')} \setminus \mathcal{I}_{\mathsf{Train}}^{(M)}$ on the other maps, namely an equal number of i.i.d. CT nodes sampled from each of the four other maps. We use only 30 training instances since they are sufficient for

	Warehouse	Room	Random	City	Game
Number of agents in MAPF instances in $\mathcal{I}_{\mathsf{Train}}$ and $\mathcal{I}_{\mathsf{Valid}}$	30	22	18	180	100
Training on the same map — Swapped pairs (%)	5.78	12.58	10.89	2.89	4.40
Training on the same map — Top pick accuracy (%)	84.93	67.56	69.03	83.05	60.16
Training on the other maps — Swapped pairs (%)	6.08	15.24	19.64	7.66	7.45
Training on the other maps — Top pick accuracy (%)	86.85	66.80	50.44	78.57	53.13

Table 2.3: Numbers of agents in MAPF instances in $\mathcal{I}_{\mathsf{Train}}$ and $\mathcal{I}_{\mathsf{Valid}}$, validation losses and accuracies. The swapped pairs are the percentages of swapped pairs averaged over all test CT nodes, and the top pick accuracy is the accuracy of the ranking function selecting one of the conflicts labeled as 1 in the test dataset.

collecting 5,000 CT nodes for each grid map. For each grid map $M \in \mathcal{M}$, we denote CBS+ML that uses the ranking function trained on the same grid map by CBS+ML-S and CBS+ML that uses the one trained on the other maps by CBS+ML-O. We set the regularization parameter $C = 1/100$ to train an SVM$^{\mathsf{rank}}$ [94] with a linear kernel to obtain each of the ranking functions. We varied $C \in \{1/10, 1/100, 1/1000\}$ and achieved similar results. We test the learned ranking functions on the validation dataset collected by solving $\mathcal{I}_{\mathsf{Valid}}^{(M)}$. The numbers of agents in the training instances used for data collection, the validation losses and the accuracies of selecting one of the conflicts labeled as 1 are reported in Table 2.3. We varied the number of agents for data collection and find that they led to similar performance. In general, the losses of the ranking functions for CBS+ML-O are larger and their accuracies of selecting "good" conflicts are lower than those for CBS+ML-S.

2.6.2.2 Results

Success Rate, Runtime and Tree Size We run CBSH2, CBS+ML-S and CBS+ML-O on 25 test instances on each of the five maps and vary the number of agents. The runtime limits are set to 60 minutes for the two largest maps (the city and game maps) and 10 minutes for the other maps. In Table 2.4, we report the success rates together with the average runtimes and the average CT sizes of the test instances solved by all methods for different numbers of agents on each grid map. CBS+ML-S and CBS+ML-O dominate CBSH2 in all metrics on all maps for almost all cases. For CBS+ML-S, even though we learn the ranking

Grid Map	k	Success Rate (%)			Runtime (min)			CT Size (nodes)		
		CBSH2	ML-S	ML-O	CBSH2	ML-S	ML-O	CBSH2	ML-S	ML-O
Warehouse	30	**100**	**100 (100)**	**100 (100)**	0.15	**0.05**	**0.05**	541	**131**	152
	36	92	**100 (92)**	**100 (92)**	0.42	**0.06**	0.07	1992	**268**	321
	42	52	**68 (52)**	**68 (52)**	1.97	0.63	**0.32**	11243	3533	**1503**
	45	28	**44 (28)**	**48 (28)**	3.41	1.33	**0.50**	17348	5320	**1997**
	48	16	**36 (16)**	**40 (16)**	0.23	**0.10**	0.12	1328	**517**	676
	54	4	**20 (4)**	**20 (4)**	0.49	1.11	**0.20**	2808	5633	**1087**
		Improvement over CBSH2			0	73.5%	**76.1%**	0	77.0%	**81.9%**
Room	22	**96**	**96 (96)**	**96 (96)**	0.08	**0.07**	**0.07**	313	228	**227**
	26	**80**	**80** (76)	**80** (76)	0.94	0.44	**0.42**	10983	4373	**3859**
	28	60	**72 (60)**	68 (**60**)	1.43	1.02	**1.01**	17551	10505	**9968**
	30	40	**44 (40)**	**44 (40)**	1.76	**0.98**	1.04	23250	**11348**	11987
	32	**24**	**24** (20)	**24** (20)	3.16	**2.26**	**2.26**	42041	27035	**25986**
	34	4	**8 (4)**	4 (**4**)	2.68	**0.67**	0.80	36137	**7925**	9090
		Improvement over CBSH2			0	**34.3%**	33.1%	0	**47.7%**	47.2%
Random	18	**92**	**92 (92)**	**92 (92)**	0.16	**0.15**	**0.15**	2609	2366	**2302**
	20	**88**	**88 (88)**	**88 (88)**	0.90	**0.88**	0.89	8779	7897	**7742**
	23	60	**72** (56)	68 (56)	1.60	1.37	**1.40**	27628	**23970**	24257
	26	**40**	**40 (40)**	**40 (40)**	3.01	2.85	**2.51**	47297	44770	**43094**
	29	24	**36 (24)**	**28 (24)**	4.44	**3.38**	3.46	64965	**52105**	52463
		Improvement over CBSH2			0	**27.4%**	25.2%	0	**25.3%**	22.8%
City	180	76	**84 (76)**	**84 (19)**	4.73	**3.37**	3.45	578	**280**	285
	200	72	**76 (72)**	**76 (72)**	7.92	**4.62**	4.76	878	**288**	297
	220	48	64 (**48**)	**68 (48)**	6.04	4.47	**3.73**	934	445	**402**
	240	36	44 (**36**)	**48 (36)**	9.25	5.91	**5.71**	790	510	**507**
	260	24	24 (**24**)	**32 (24)**	10.59	8.80	**8.62**	1363	1088	**1074**
	280	24	24 (**24**)	**28 (24)**	12.72	14.27	**10.16**	1529	1650	**1414**
		Improvement over CBSH2			0	23.6%	**29.8%**	0	40.9%	**42.4%**
Game	100	72	**76 (72)**	**76 (72)**	6.47	4.88	**3.87**	3418	1729	**1470**
	110	44	**56 (44)**	**52 (44)**	7.13	4.47	**4.44**	3157	**1312**	1366
	115	44	**48 (44)**	**48 (44)**	8.06	4.96	**4.95**	3990	**1753**	1805
	120	**36**	**36 (36)**	**36 (36)**	12.48	6.59	**6.47**	6176	3664	**3536**
	130	24	**32 (24)**	28 (24)	14.03	16.52	**13.89**	6649	6700	**6621**
	135	20	**28 (20)**	24 (20)	8.88	11.68	**8.43**	2537	2598	**2502**
		Improvement over CBSH2			0	27.2%	**37.6%**	0	45.7%	**48.7%**

Table 2.4: Success rates and the average runtimes and CT sizes of MAPF instances solved by all methods (ML-S and ML-O stand for CBS+ML-S and CBS+ML-O, respectively) for different numbers of agents k on five maps. For the success rates of ML-S and ML-O, the percentages of MAPF instances solved by both our methods and CBSH2 are given in parentheses (bolded if they solve all MAPF instances that CBSH2 solves). For each grid map, we report the percentages of our improvement over CBSH2 on the runtime and CT size on MAPF instances solved by all methods.

function from data collected on instances with a fixed number of agents (listed in Table 2.3), the learned function generalizes to instances with larger numbers of agents on the same grid map and outperforms CBSH2. CBS+ML-O, without seeing the actual grid map being tested on during training, is competitive with CBS+ML-S. The results suggest that our approach, when focusing on solving instances on a particular grid map, can outperform CBSH2 substantially and, when faced with new maps, still has an advantage. CBS+ML-O even outperforms CBS+ML-S sometimes on the warehouse, city, and game maps (similarly in Table 2.3), which suggests that learning from the expert's demonstration on multiple grid maps benefits CBS+ML since the effectiveness of the expert varies across grid maps.

Feature Importance   Next, we look at the feature importance of the learned ranking functions. For CBS+ML-O, the five ranking functions have nine features in common among their eleven features with the largest absolute weights. Thus, they are similar when looking at the important features. We take the average of each weight and sort them in decreasing order of their absolute values. The top eight features are (1) the weight of the edge between agents a_i and a_j in the WDG; (2) the binary indicator for non-cardinal conflicts; (3) the maximum of the differences of the cost of the path of agent a_i (a_j) and t; (4) the binary indicator for cardinal conflicts; (5) the minimum of the numbers of conflicts that agent a_i (a_j) is involved in; and (6-8) the minimum, the maximum and the sum of the numbers of conflicts involving agent a_i (a_j) that have been selected and resolved. Those features mainly belong to three categories: features related to the conflict type, the WDG and the number of conflicts having been resolved for agents, where the first one is commonly used in previous work on CBS and the third one is an analogue of the branching variable pseudocosts in Branch-and-Bound for MILP solving [2].

2.7 Learning to Select Nodes for ECBS

In this section, we introduce ECBS+ML to show how the framework to learn heuristic can be applied to improve node selection in Enhanced Conflict-Based Search (ECBS) [174]. MAPF is NP-hard to solve optimally [209, 9] and, therefore, optimal MAPF search algorithms, such as CBS, do not scale to many agents. ECBS and its variants [35, 36] are guaranteed to find solutions whose sums of costs of the paths are at most $w \geq 1$ times the minimum ones and run faster than optimal MAPF search algorithms. Which CT nodes to select from the focal list for expansion is important decision-making in ECBS. A generic node-selection strategy for ECBS assigns a d-value to each CT node and always selects a CT node with the minimum d-value in the focal list for expansion. The most commonly used node-selection strategy in previous work uses the number of conflicts $|N_{\mathsf{Conf}}|$ as the d-value of a CT node N. We refer to this strategy as h_1. [10] propose other strategies that use the number of pairs of agents that have at least one conflict with each other and the number of agents that have at least one conflict with other agents as the d-values. We refer to these two strategies as h_2 and h_3, respectively. We implement and experiment with h_1, h_2 and h_3 in Section 2.7.2.

Instead of manually defining the d-values, we introduce ECBS+ML to show how our framework can be applied to improving node selection in ECBS. We borrow tools such as imitation learning and curriculum learning from the machine learning literature [167, 168, 178, 15] and propose a novel method for learning node-selection strategies for the high-level focal search to speed up ECBS. This method could also be applied to the low-level focal search but we focus on the high-level one since the low-level focal search runs in polynomial time and does not have as big an impact on the efficiency of ECBS as the high-level one. We then empirically demonstrate the effectiveness and efficiency of ECBS+ML.

2.7.1 Machine Learning Methodology

For training, we do not directly learn the d-values but rather a ranking function that differentiates CT nodes that have shorter distances to leaf nodes in the CT with w-approximate solutions from those CT nodes that have longer distances to one. During the search, the ranking function takes a CT node's features as input and calculates a real-valued d-value. Our goal is to learn a ranking function such that its d-values allow ECBS to get closer to a w-approximate solution every time it expands a CT node and, therefore, help it to find a solution more quickly. The node-selection strategies are applied when selecting a node to expand the CT. Thus, we represent the state of the search with the CT $s = \mathcal{T}'$ ECBS built prior to selecting the node. ECBS can select any CT node from the focal list $\mathcal{F}(T')$ when T' is built, thus, the available actions are all CT nodes in the list, i.e., $A(\mathcal{T}') = \mathcal{F}(T')$. To apply the framework, we first propose an expert that retrospectively computes the complete CT to select CT nodes. The complete CT is defined as the CT we get when ECBS terminates, which is expensive to compute. Next, given the expert, we explain how we use it to collect data and apply ML to imitate its decisions. During training, we fix the underlying graph and learn node-selection strategies from solving the training instances on that graph where the start and goal vertices of the agents are drawn from a given distribution. We start with a small number of agents and use imitation learning [40, 167, 168] to learn a node-selection strategy for that number of agents. We then continue learning node-selection strategies for larger and larger numbers of agents. Instead of learning from scratch every time the number of agents increases, we use curriculum learning [15] to learn more efficiently by using previously learned node-selection strategies as starting points.

2.7.1.1 Expert for Node Selection

Given a MAPF instance, an expert for node selection at a given state $\mathcal{T}'$ is a ranking function π' that takes the focal list $\mathcal{F}(\mathcal{T}')$ as input, calculates a real-valued score per CT node $N \in \mathcal{F}(\mathcal{T}')$ as its d-value and

outputs the ranks determined by the scores. We say that ECBS follows an expert for node selection iff ECBS builds the CT by always selecting the CT node with the highest rank to expand.

The expert we propose retrospectively computes the complete CT to find all w-approximate solutions to the MAPF instance. For each CT node $N \in \mathcal{F}(\mathcal{T}')$, we define N_d as the distance between N and any w-approximate solution found within the subtree rooted at N. We assign $N_d = \infty$ if no solution was found within its subtree. The expert outputs the ranks determined by the increasing order of the d-values (i.e., the highest rank for the smallest d-value). However, retrospectively computing the complete CT and finding all w-approximate solutions are prohibitively expensive computationally. Therefore, in practice, we run the expert until either it exceeds a runtime limit, T w-approximate solutions have been found or the expert terminates.

2.7.1.2 Data Collection

Given an instance I and the expert's ranking function π', we describe a subroutine CollectData(I, π') for data collection that is used in our learning algorithm. CollectData(I, π') runs ECBS following the expert using the node-selection strategy π' and returns the complete CT $\mathcal{T}$.

Features For each CT node $N \in \mathcal{F}$ at state $\mathcal{T}'$, it computes the following atomic features $f_1, \ldots, f_9$:

1. features related to the conflicts: the number of conflicts $|N_{\mathsf{Conf}}|$ (f_1), the number of pairs of agents that have at least one conflict with each other (f_2) and the number of agents that have at least one conflict with other agents (f_3);

2. features related to N_{Cost}: $f_4 := N_{\mathsf{Cost}}, f_5 := \frac{N_{\mathsf{Cost}}}{\mathsf{LB}}, f_6 := N_{\mathsf{Cost}} - \mathsf{LB}, f_7 := N_{\mathsf{Cost}} - S$ and $f_8 := N_{\mathsf{Cost}}/S$, where S is the sum of costs of the individually cost-minimal paths of all agents; and

3. the depth of N in the CT (f_9).

From these atomic features, we obtain interaction features $f_i f_j$ (for $i \leq j$), which are the pairwise products of the atomic features. The final feature vector $\phi_{\mathcal{T}'}(N) \in \mathbb{R}^p$ ($p = 54$) is obtained by concatenating all atomic features and interaction features, resulting in the degree-2 polynomial kernel in the space of atomic features. Features f_2 and f_3 can be computed in time $O(|N_{\mathsf{Conf}}|)$, and the other features can be computed in time $O(1)$. Therefore, the overall time complexity for computing all 54 features is $O(|N_{\mathsf{Conf}}|)$.

The interaction features are a richer set of features than the atomic features. This is also known as using the polynomial kernel in SVM, where we use a polynomial with degree two. But using them increases the runtime for training an SVM$^{\mathsf{rank}}$. It is not an issue for learning to select nodes in ECBS since we have only nine atomic features, in contrast to having 67 of them in the previous section where we learn to select which conflict to resolve next in CBS. We do not use interaction features either in Sections 2.8 and 2.9 for the same reasons.

Labels During data collection, we run the expert until either T solutions are found, the search exceeds the runtime limit or the expert terminates. If the expert exceeds the runtime limit without finding any solution, we return an empty set of training data for instance I. Otherwise, we return the set of states encountered during the search. We assign a label $y_{\mathcal{T}'}(N)$ to each CT node $N \in \mathcal{F}(T')$ based on the minimum distance N_d between N and any w-approximate solution found within the subtree rooted at N. We assign $N_d = \infty$ if no solution was found within its subtree. Since we want to assign smaller d-values

to CT nodes that are closer to a solution, we label N in a way such that the closer N is to a solution, the smaller y_N is:

$$y_{\mathcal{T}'}(N) = \begin{cases} 0, & \text{if } N_d < \tau_0, \\ 1, & \text{if } \tau_0 \leq N_d < \tau_1, \\ 2, & \text{if } \tau_1 \leq N_d < \tau_2, \\ 3, & \text{if } \tau_2 \leq N_d < \infty, \\ \infty, & \text{otherwise,} \end{cases}$$

where $\tau_2 > \tau_1 > \tau_0 > 0$ are three thresholds. Our labeling scheme allows us to focus on pairs of CT nodes that have large differences in N_d when the labels are used to learn a ranking function. Different from using $y_{\mathcal{T}'}(N) = N_d$, it avoids having to rank CT nodes correctly that are almost equally good or bad, which is irrelevant for making good node selections. We indeed tried using $y_{\mathcal{T}'}(N) = N_d$ but did not get better performance than ECBS without ML-guided node selection.

2.7.1.3 Model Learning

We want to learn node-selection strategies for instances with different numbers of agents on a fixed underlying graph $G = (V, E)$ with a fixed suboptimality factor w. The idea central to our training algorithm is that we start learning a node-selection strategy by solving easy instances with a small number of agents and iteratively increasing the number of agents to learn another strategy based on the previous one. In particular, we want to learn to solve instances with increasing difficulty, i.e., with m different numbers of agents $k_1, \ldots, k_m$ where $k_1 < \ldots < k_m$. For each k_i, we learn a node-selection strategy that assigns $\pi_i(\phi_{\mathcal{T}'}(N))$ to CT node N as its d-value, where π_i is a learned ranking function. Therefore, a desirable ranking function is one that assigns smaller d-values to CT nodes that are closer to a w-approximate solution and larger d-values to those CT nodes that are farther away from one.

Algorithm 3 Training Algorithm: Curriculum Learning

1: **Input:** $\{k_1, \ldots, k_m\}$ and m sets of training instances $\{\mathcal{I}_1, \ldots, \mathcal{I}_m\}$
2: $\pi_0 \leftarrow \pi^*$
3: **for** $i = 1$ to m **do**
4:     $\pi_i \leftarrow \text{DAgger}(\pi_{i-1}, \mathcal{I}_i)$ $\triangleright$ Call Algorithm 4
5:     **if** $\pi_i = \pi_{i-1}$ **then** $\triangleright$ Stopping criterion met
6:         $\forall i < j \leq m, \pi_j \leftarrow \pi_i$
7:         **break**
8: **return** $\{\pi_1, \ldots, \pi_m\}$

One of the main challenges is that, as k_i increases, it becomes increasingly hard to collect a sufficient amount of training data due to the increased difficulties of the MAPF instances and, thus, the increased runtime to collect data. To overcome this challenge, we propose a training algorithm based on curriculum learning, as shown in Algorithm 3. Curriculum learning is a machine learning technique that trains an ML model using examples (in our case, MAPF instances) of increasing difficulty. The ML model is first trained on simple tasks (in our case, node selection in ECBS on easier MAPF instances) and then knowledge from those tasks is transferred to the difficult task. Algorithm 3 takes $\{k_1, \ldots, k_m\}$ and m sets of training instances $\{\mathcal{I}_1, \ldots, \mathcal{I}_m\}$ as input and outputs $\{\pi_1, \ldots, \pi_m\}$. Each instance in $\mathcal{I}_i$ includes k_i agents, where the start and goal vertices of the agents are drawn i.i.d. from a given distribution. π_0 is set to a ranking function π^* that corresponds to an initial node-selection strategy (e.g., one of node-selection strategies h_1, h_2 and h_3) (Line 2). To obtain π_1 for instances with k_1 agents, we use $\text{DAgger}(\pi^*, \mathcal{I}_1)$ [168] (see Algorithm 4) as a training algorithm that learns a ranking function from solving the training instances in $\mathcal{I}_1$ using π^* as a starting point. To obtain π_i (for $i > 1$), instead of starting from π^* again, we start learning from π_{i-1}. We obtain π_i ($1 < i \leq m$) by calling $\text{DAgger}(\pi_{i-1}, \mathcal{I}_i)$, which learns a ranking function starting from π_{i-1} as the ranking function of the node-selection strategy (Line 3-4) until a stopping criterion is met (Lines 5-7) or $i > m$. If the stopping criterion is met before $i > m$, we terminate training (Line 7) and simply set $\pi_j = \pi_i$ for all $i < j \leq m$ (Line 6). If $\text{DAgger}(\pi_{i-1}, \mathcal{I}_i)$ returns π_{i-1}, then it cannot find a better ranking function than π_{i-1}. This situation typically occurs at some point in time during training for hard instances with many agents since only data collected from solved instances during data collection contributes to the

training data, and, for hard instances, it is difficult to collect a sufficient amount of training data, which makes it difficult to improve on π_{i-1}. When the training algorithm observes this situation (Line 5), it stops training and uses the last obtained ranking function for all instances with larger numbers of agents than the one for which it could not improve the ranking function.

DAgger$(\pi^{(0)}, \mathcal{I}_i)$, shown in Algorithm 4, is an imitation learning algorithm. The inputs $\pi^{(0)}$ and $\mathcal{I}_i$ are the ranking function of the initial node-selection strategy and the set of training instances, respectively. DAgger repeatedly determines a ranking function that makes better decisions in those situations that were encountered when running ECBS with the previous version of the ranking function. Initially, the training data D is set to $\emptyset$ (Line 1). Let R be the number of iterations for which the algorithm runs (Line 4). In iteration j, it collects training data by solving the instances in $\mathcal{I}_i$ with the ranking function $\pi^{(j-1)}$ obtained in iteration $j-1$, aggregates it with D (Line 6) and learns a new ranking function $\pi^{(j)}$ from D that minimizes a loss function over D (Line 7). When collecting training data using $\pi^{(j-1)}$ in ECBS, we set a runtime limit for each instance. We record the success rate (i.e., the fraction of instances solved within the given runtime limit) on $\mathcal{I}_i$ (Line 8) and the average runtime for the solved instances in $\mathcal{I}_i$ (Line 9). Finally, DAgger returns the ranking function that achieves the highest success rate on the instances in $\mathcal{I}_i$ in all R iterations (Line 10), breaking ties in favor of the lowest average runtime for the solved instances (Line 11).

Learning a Ranking Function One could follow the learning-to-rank formulation in Section 2.5 to train a ranking function that minimizes the loss function $l(y_{\mathcal{T}'}, \hat{y}_{\mathcal{T}'})$ across all states. However, $l(y_{\mathcal{T}'}, \hat{y}_{\mathcal{T}'})$ needs to consider $\mathcal{P}_{\mathcal{T}'}$ which consists of all ordered CT node pairs in the focal list $\mathcal{F}(\mathcal{T}')$ and has a quadratic number of CT node pairs. Thus, the total number of CT node pairs needed to be considered to compute the loss function for a single CT $\mathcal{T}$ is $O(|\mathcal{T}|^3)$ for just a single training instance where $|\mathcal{T}|$ is the number of CT nodes in $\mathcal{T}$, and it would be prohibitively expensive to compute.

Algorithm 4 DAgger($\pi^{(0)}, \mathcal{I}_i$)

1: $D = \emptyset$
2: $r_0 \leftarrow$ success rate on $\mathcal{I}_i$ using $\pi^{(0)}$ in ECBS
3: $c_0 \leftarrow$ average runtime on solved instances in $\mathcal{I}_i$ using $\pi^{(0)}$ in ECBS
4: **for** $j = 1$ to R **do**
5:      **for** I in training instance set $\mathcal{I}_i$ **do**
6:          $D \leftarrow D \cup \text{CollectData}(I, \pi^{(j-1)})$ ▷ Call ECBS
7:      $\pi^{(j)} \leftarrow$ train a ranking function using D
8:      $r_j \leftarrow$ success rate on $\mathcal{I}_i$ using $\pi^{(j)}$ in ECBS
9:      $c_j \leftarrow$ average runtime on solved instances in $\mathcal{I}_i$ using $\pi^{(j)}$ in ECBS
10: $L \leftarrow \arg\max_{0 \leq l' \leq R}\{r_{l'}\}$
11: $l \leftarrow$ an element from $\arg\min_{l' \in L}\{c_{l'}\}$
12: **return** $\pi^{(l)}$

Notice that the value of label $y_{\mathcal{T}'}(N)$ depends solely on the complete CT $\mathcal{T}$, i.e., $y_{\mathcal{T}'}(N) = y_{\mathcal{T}}(N)$ for any $\mathcal{T}'$. Also, notice that the features $\phi_{\mathcal{T}'}(N)$ depend only on the information of CT node N and, thus, $\phi_{\mathcal{T}'}(N) = \phi_{\mathcal{T}}(N)$ for any $\mathcal{T}'$.

Based on the above observation, we propose to consolidate the states $\mathcal{T}'$ of a MAPF instance to a single state represented by the complete CT $\mathcal{T}$ and call $\mathcal{T}' \subseteq \mathcal{T}$ a substate of $\mathcal{T}$. To compute the loss for the consolidated state $\mathcal{T}$, we consider all ordered CT node pairs in every substate, i.e., we let

$$\tilde{\mathcal{P}}_{\mathcal{T}} = \bigcup_{\mathcal{T}' \subseteq \mathcal{T}} \mathcal{P}_{\mathcal{T}'}.$$

$\tilde{\mathcal{P}}_{\mathcal{T}}$ consists of all ordered pairs of CT nodes that occur in the focal list during the search. After the consolidation, the training dataset D is a set of complete CTs, one for each training instance. We train a linear ranking function with parameter $\boldsymbol{w} \in \mathbb{R}^p$

$$\pi : \mathbb{R}^p \to \mathbb{R} : \pi(\phi_{\mathcal{T}}(N)) = \boldsymbol{w}^{\mathsf{T}} \phi_{\mathcal{T}}(N)$$

and minimize the loss function

$$L(\boldsymbol{w}) = \sum_{\mathcal{T} \in D} l(y_{\mathcal{T}}, \hat{y}_{\mathcal{T}}) + \frac{C}{2} ||\boldsymbol{w}||_2^2$$

over the training data D, where $y_{\mathcal{T}}$ is the ground-truth label vector of all CT nodes that appear in $\mathcal{T}$, $\hat{y}_{\mathcal{T}}$ is the corresponding vector of predicted values resulting from applying π to the feature vector $\phi_{\mathcal{T}}(N)$, and $l(y_{\mathcal{T}}, \hat{y}_{\mathcal{T}})$ is computed as follows:

$$l(y_{\mathcal{T}}, \hat{y}_{\mathcal{T}}) = \frac{\sum_{(N_i, N_j) \in \tilde{\mathcal{P}}_{\mathcal{T}} : \hat{y}_{\mathcal{T}}(N_i) \leq \hat{y}_{\mathcal{T}}(N_j)} \tilde{w}_{N_i, N_j}}{\sum_{(N_i, N_j) \in \tilde{\mathcal{P}}_{\mathcal{T}}} \tilde{w}_{N_i, N_j}}. \tag{2.1}$$

The weight $\tilde{w}_{N_i, N_j}$ of each pair $(N_i, N_j) \in \tilde{\mathcal{P}}_{\mathcal{T}}$ is set to $e^{-(d_i + d_j)/rd_{\max}}$, where d_i and d_j are the depths of N_i and N_j in $\mathcal{T}$ respectively, $d_{\max}$ is the depth of $\mathcal{T}$, and r is a damping factor. The weight $\tilde{w}_{N_i, N_j}$ takes into account the fact that the CT grows exponentially and can be understood as the product of the weights of N_i and N_j, where the weight of N_i is $e^{-d_i/rd_{\max}}$. We use the weighted version of the ranking loss to help focus on making accurate predictions for CT nodes that are close to the root early in the search since expanding a CT node that contains no $w-$approximate solution would bring an extra computation cost that is exponential of the depth of the sub-CT rooted at that CT node in the worst case. Alternatively, one could argue setting the weight $e^{-\min(d_i, d_j)/rd_{\max}}$ since the importance of ranking a pair of CT nodes (N_i, N_j) correctly depends on the closest solution to either of N_i or N_j. We did not try this in our empirical evaluation but do not want to rule out the possibility that this could also be a good choice for setting the weights.

2.7.1.4   ML-Guided Search

After learning the ranking functions $\{\pi_1, \ldots, \pi_m\}$, we deploy them in ECBS. Given an instance with k agents and the same underlying graph as used during training, we run ECBS with ranking function π_j,

where $j \in \arg\min_{i \in [m]}\{|k - k_i|\}$, i.e. the one trained on the most similar number of agents. When a CT node N is generated, we compute its feature vector $\phi_{\mathcal{T}}(N)$ and set its d-value to $\pi_j(\phi_{\mathcal{T}}(N))$. The overall time complexity of computing the d-value is $O(|N_{\mathsf{Conf}}|)$ because of the time complexity of computing the features. Even though the time complexity of computing the d-value for node-selection strategy h_1 is $O(1)$, we will show experimentally that ECBS+ML outperforms ECBS with h_1 in terms of both the success rate and the runtime.

Discussion   Our main motivation to train multiple ranking functions for different numbers of agents is that, as will be shown in Section 5.2, the ranking functions learned with $\mathrm{DAgger}(\cdot, \cdot)$ do not generalize well to instances with different numbers of agents, especially when those numbers are substantially larger than the one we train on. There are two reasons for this issue: (1) We are not able to normalize the feature values based on their minimums and maximums as we did for CBS+ML since we need to compute the d-value of a CT node immediately during the search when it is generated, but the minimum and maximum are not known until ECBS terminates. (2) Different features are important for instances with different numbers of agents, which will be shown in Section 2.7.2. Therefore, we learn node-selection strategies specific to the number of agents, and we use curriculum learning to learn them efficiently. In contrast, we did not use curriculum learning in Section 2.6 since we observed that the ML models generalized well to MAPF instances with larger numbers of agents than the training instances. However, it is possible that curriculum learning can further improve the performance of CBS+ML.

There are heuristic components in our training algorithm. The first component is the design of the stopping criterion for curriculum learning (Line 5 in Algorithm 3). If we cannot improve on π_{i-1} in iteration i of the training algorithm, we are not able to improve on it in subsequent iterations either. The other component is the criterion for choosing the best ranking function in DAgger. One could argue that the best ranking function should be chosen based on the performance on a set of validation instances drawn from the same distribution as for training (Lines 8-9 in Algorithm 4). However, we do not use validation

Grid Map	Random	Warehouse	Maze	Game	City
w	1.1	1.05	1.01	1.005	1.005
m	10	11	9	16	13
k_1	75	140	45	80	160
k_m	125	240	125	305	400
$|V|$	819	5,699	14,818	28,178	47,240

Table 2.5: Parameters for each grid map. w is the suboptimality factor, m is the number of different numbers of agents we train and test on, k_1 is the smallest number of agents that we train and test on, k_m is the largest number of agents that we train and test on, and $|V|$ is the number of unblocked cells on the grid map. $k_2, \cdots, k_{m-1}$ are evenly distributed on $[k_1, k_m]$, i.e., $k_i = (i-1)(k_m - k_1)/(m-1) + k_1$.

Grid Map	Random	Warehouse	Maze	Game	City
l_1	0.0075	0.0088	0.0092	0.0085	0.0051
$l_{\lfloor m/2 \rfloor}$	0.0330	0.0166	0.0192	0.0131	0.0068
l_m	0.0653	0.0283	0.0318	0.0204	0.0107

Table 2.6: Loss $l_i \in [0, 1]$ of ranking function π_i for k_i agents evaluated by Equation (2.1) averaged over all CTs in the training data.

instances since this would approximately double the runtime of DAgger and, thus, also of the training algorithm if the numbers of instances for validation and training were the same. Our criterion allows the training algorithm to select a good ranking function more efficiently.

We learn the ranking function differently from Section 2.6 in several aspects. First, we use a weighted version of the loss function since the unweighted version did not perform well on all the grip maps except the random map that we tested in our empirical evaluation in the next subsection. Secondly, we consolidate the training dataset by considering the pairs of CT nodes that occur in multiple $\mathcal{P}_{\mathcal{T}'}$ only once in the loss function to reduce the runtime for learning the ranking function. We did not do the same thing in Section 2.6 since the number of pairs of actions (i.e., conflicts at a CT node) for conflict selection in CBS was much smaller than the number of pairs of actions (i.e., CT nodes in the focal list) for node selection in ECBS.

2.7.2 Empirical Evaluation

In this subsection, we demonstrate the efficiency of ECBS+ML through experiments. In the following, we introduce our evaluation setup and then present the results.

2.7.2.1 Setup

We implement ECBS+ML in C++ and conduct our experiments on a 2.4 GHz Intel Core i7 CPU with 16 GB RAM. During testing, we compare against ECBS with the node-selection strategies h_1, h_2 and h_3, denoted by ECBS+h_1, ECBS+h_2 and ECBS+h_3, respectively. We also compare against two versions of ECBS+ML, one that stops early, denoted by ECBS+ML(ES), and one that uses only imitation learning without curriculum learning, denoted by ECBS+IL. In our ablation study, we also compare against ECBS+ML(ES) and ECBS+IL. ECBS+ML(ES) uses the same training algorithm as ECBS+ML except that it stops training earlier than ECBS+ML. The number of agents in the last iteration of training in the training algorithm is the one where the success rate of ECBS+h_1 first drops below 60%. ECBS+IL uses the same training algorithm as ECBS+ML except that, for each number of agents, it learns a ranking function starting from the given initial ranking function without relying on the previously learned one. We replace Line 4 in Algorithm 3 with "$\pi_i \leftarrow \mathrm{DAgger}(\pi^*, \mathcal{I}_i)$" and "$\pi_i = \pi_{i-1}$" on Line 5 with "$\pi_i = \pi^*$". We set the runtime limit to 5 minutes per instance for running ECBS for both data collection and testing. The number of solutions T collected during data collection is set to 10. The thresholds that determine the labels τ_0, τ_1 and τ_2 are set to 10, 30 and 60, respectively. The number of iterations R for DAgger is set to 10. The damping factor r for weight $\tilde{w}_{N_i, N_j}$ is set to 0.3727. r is chosen so that a CT node at depth $0.6d_{\max}$ has weight $e^{-0.6/r} = 0.2$. Since we are using a pairwise loss, we suffer from a quadratic time complexity ($O(|\mathcal{T}|^2)$) for the loss computation. Therefore, we record only the first 10,000 CT nodes generated for each instance during data collection. We use the default values for all parameters in LIBLINEAR, including the regularization parameter C, which is set to 1. We did not try out many other values for the hyperparameters since the improvement of ECBS+ML over ECBS is already substantial with these values.

We evaluate ECBS+ML on five grid maps of different sizes and structures from the MAPF benchmark [181], including (1) a random map "random-32-32-20", which is a 32×32 grid map with 20% randomly blocked cells; (2) a warehouse map "warehouse-10-20-10-2-1", which is a 163×63 grid map with 200

10×2 rectangular obstacles; (3) a maze map "maze-128-128-10", which is a 128×128 grid map with ten-cell-wide corridors; (4) a game map "den520d", which is a 257×256 grid map from the video game *Dragon Age: Origins*; and (5) a city map "Paris_1_256", which is a 256×256 grid map of Paris. Since ECBS has better scalability than CBS, compared to the MAPF instances used in Section 2.6.2, we use MAPF instances with larger sizes and higher obstacle and agent densities. For example, we increase the sizes of the random map and the warehouse map. We also increase the number of agents in MAPF instances on the same city map. We use 25 MAPF instances for both training and testing. The parameters related to each grid map are listed in Table 2.5. k_1 is chosen such that at least one of ECBS+h_i (i=1,2,3) has a success rate of 88% or higher. We fix the increment between k_i and k_{i+1} for each grid map, and k_m is chosen such that either the success rate of ECBS+ML falls below 20% or all ECBS+h_i (i=1,2,3) have 0% success rates. We fix the suboptimality factor w, following the reasoning in previous work [10], where small w values are chosen for large grid maps and large w values are chosen for small grid maps. Our objective is to obtain a ranking function π_i for each number of agents k_i in $\{k_1, \ldots, k_m\}$. The training loss of the learned ranking functions is shown in Table 2.6. It is small. We now test the node-selection strategies that correspond to those ranking functions on unseen instances with $k_1, \ldots, k_m$ agents.

2.7.2.2 Results

Success Rate and Runtime Figure 2.1 plots the success rates on all grid maps. Overall, ECBS+ML substantially outperforms the three baselines, ECBS+h_1, ECBS+h_2 and ECBS+h_3, on all grid maps. On the game map, in particular, the success rates of the baselines drop below 20% when the number of agents increases to 170, and the baselines can hardly solve instances with more than 245 agents, while the success rates of ECBS+ML stay above 76% for up to 245 agents and ECBS+ML can still solve 16% of the instances with 305 agents. Overall, the success rates of ECBS+ML are 52% to 80% when those of the baselines begin to drop below 20%. When the success rates of the baselines are all below 8%, ECBS+ML can still solve
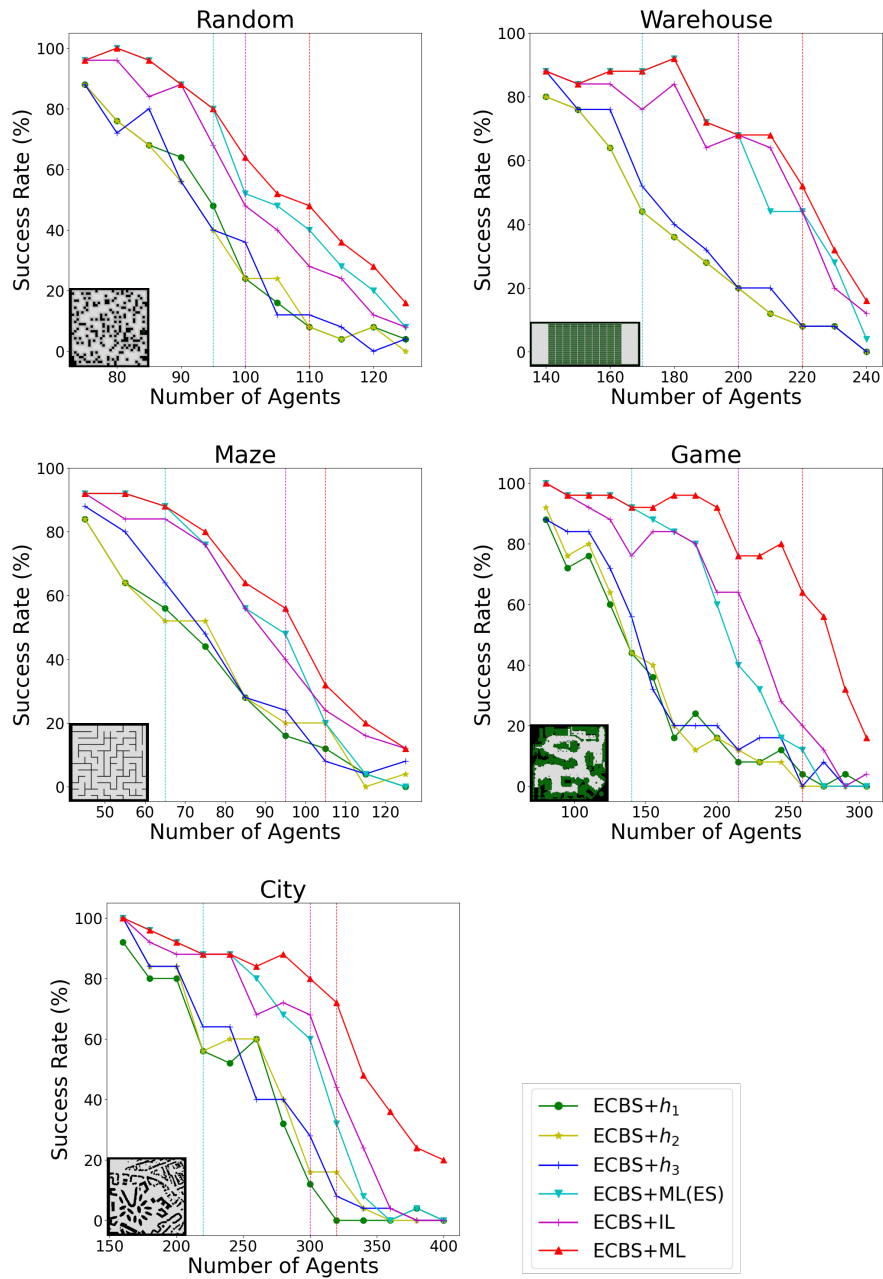
Figure 2.1: Success rates for a runtime limit of 5 minutes as a function of the number of agents for each grid map. The values of w and the numbers of agents are listed in Table 2.5. For ECBS+ML, ECBS+ML(ES) and ECBS+IL, the vertical line of the same color indicates the number of agents in the last iteration where a ranking function is learned in the training algorithm. In the figure for the warehouse map, the graph of ECBS+h_1 coincides entirely with the one of ECBS+h_2.
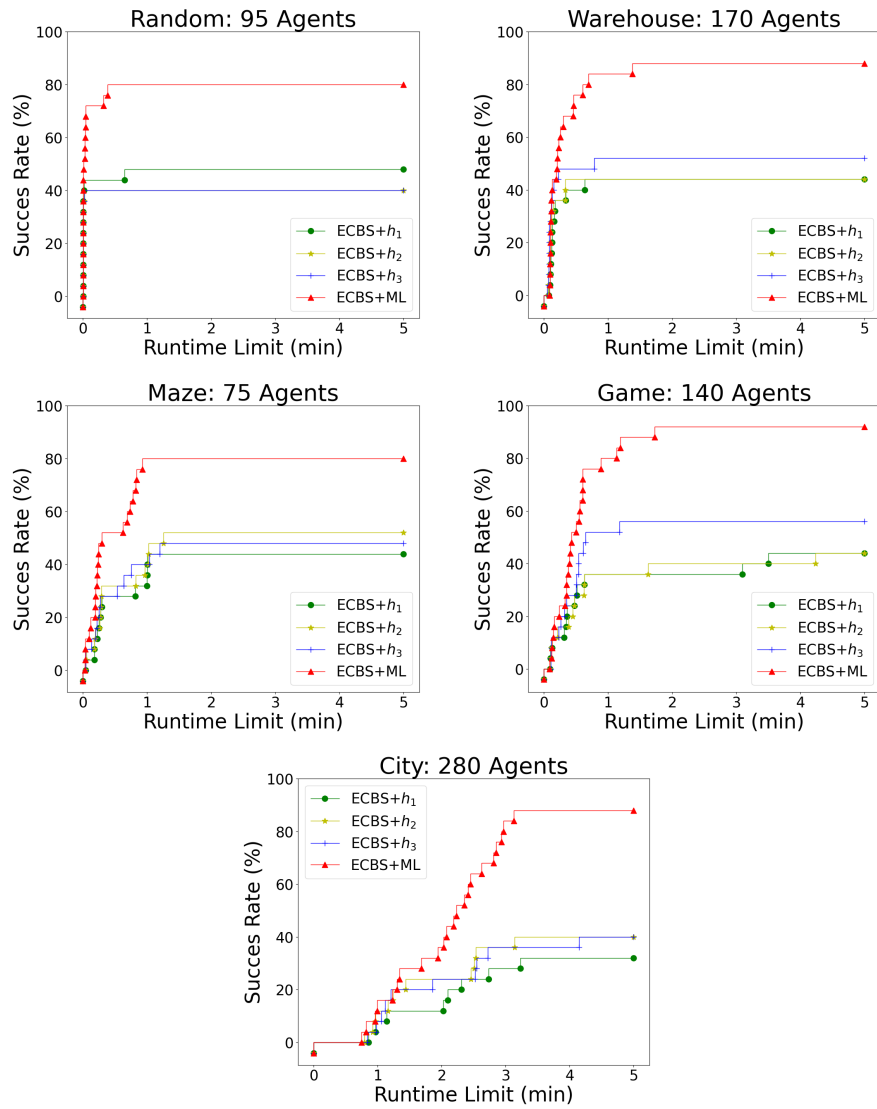
Figure 2.2: Success rates for a fixed number of agents as a function of the runtime limit for each grid map.
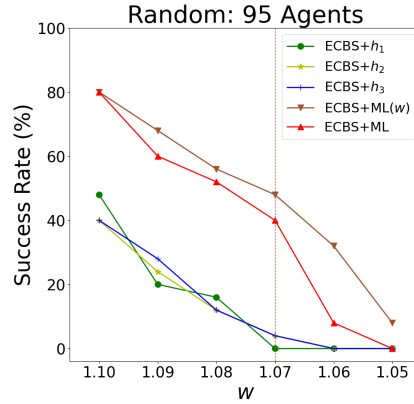
Figure 2.3: Success rates for a runtime limit of 5 minutes as a function of the suboptimality factor w on the random map for 95 agents. The vertical brown line indicates the value of w in the last iteration where a ranking function is learned for ECBS+ML(w).

instances with 9% to 17% more agents with success rates around 12% to 20%. To demonstrate the efficiency of ECBS+ML further, we show the success rates for different runtime limits in Figure 2.2. We show one figure for each grid map with a fixed number of agents, namely the smallest number of agents k_i where the baseline with the weakest heuristic has a success rate below 50% for a runtime limit of 5 minutes. In these cases, ECBS+ML has a success rate above 80% and still substantially outperforms the baselines for shorter runtime limits, e.g., of 1 or 2 minutes.

We have applied curriculum learning (Algorithm 3) to learn node-selection strategies for MAPF instances with increasing difficulties in terms of the number of agents. Next, we demonstrate that we can do the same for other measurements of difficulties, such as the suboptimality factor w. When w decreases, the difficulty increases. Figure 2.3 shows the success rates of ECBS+ML and the baselines for a runtime limit of 5 minutes as a function of the suboptimality factor w on the random map for 95 agents. We use ECBS+ML with the ranking function obtained in the experiment described in the previous paragraph that is trained on 95 agents and $w = 1.1$. To show that the success rates of ECBS+ML can be improved with curriculum learning, we use the same training algorithm (Algorithm 3) but, instead of using a fixed value for w and different numbers of agents, we use a fixed number of agents and different values of w, namely,

$w_i \in \{1.09, 1.08, \ldots, 1.05\}$. We then obtain a ranking function for each w_i. Figure 2.3 shows the success rates of the resulting variant ECBS+ML(w). ECBS+ML(w) achieves higher success rates by applying curriculum learning on different values of w than ECBS+ML, which just generalizes the ranking function for $w = 1.1$ to other values of w.

Ablation Analysis   To assess the effect of curriculum learning, we perform two ablation analyses. First, we experiment with ECBS+ML(ES). The success rates of ECBS+ML(ES) are shown in Figure 2.1. ECBS+ML(ES) is competitive with ECBS+ML on the random, warehouse and maze maps and outperforms all baselines on the random and warehouse maps, but its success rates on the city and game map drop dramatically beyond the number of agents that ECBS+ML(ES) stopped training at. The results imply that the learned node-selection strategy does not generalize well to larger numbers of agents on some grid maps and curriculum learning helps to learn better strategies in those cases.

Second, we experiment with ECBS+IL. The success rates of ECBS+IL are shown in Figure 2.1. ECBS+IL outperforms the baselines but not as substantially as ECBS+ML. The results show another two advantages of curriculum learning: (1) It enables learning for one to three more iterations (see the gaps between the vertical lines for ECBS+ML and ECBS+IL in Figure 2.1) than ECBS+IL by enabling DAgger to collect more data for training due to being provided with better node-selection strategies for this purpose; and (2) it obtains better node-selection strategies based on the previously-learned strategies than ECBS+IL that learns the node-selection strategy from the given initial ranking function in every iteration.

Feature Importance   Next, we study the feature importance of the learned ranking functions of ECBS+ML, measured by the permutation feature importance [4] of each feature, which is the increase in the loss on the training data after randomly permuting the values of that feature across all CT nodes for each CT in the training data. In Figure 2.4, we plot the normalized permutation feature importance of the top 12 features of the ranking functions for some numbers of agents and some grid maps. We first study the important
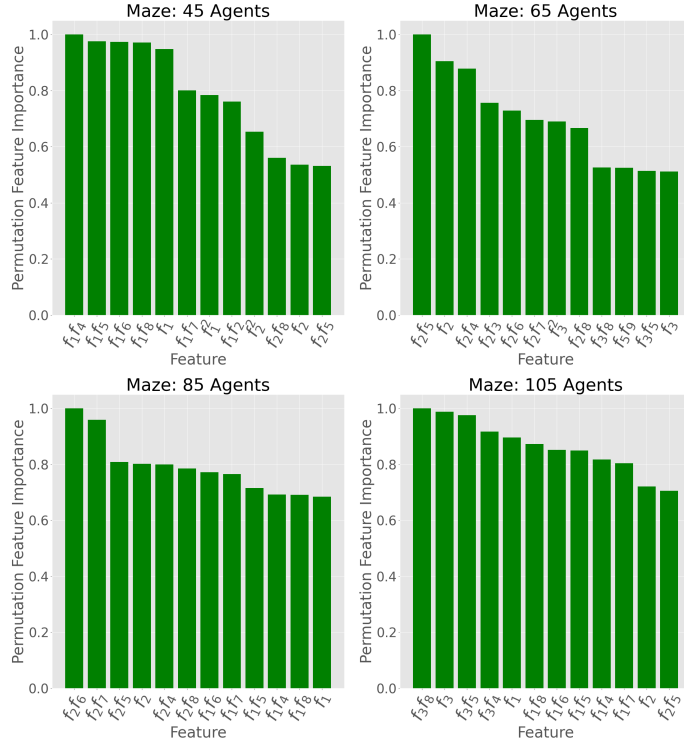
(a) Permutation feature importance of the learned ranking functions for different numbers of agents on the maze map.
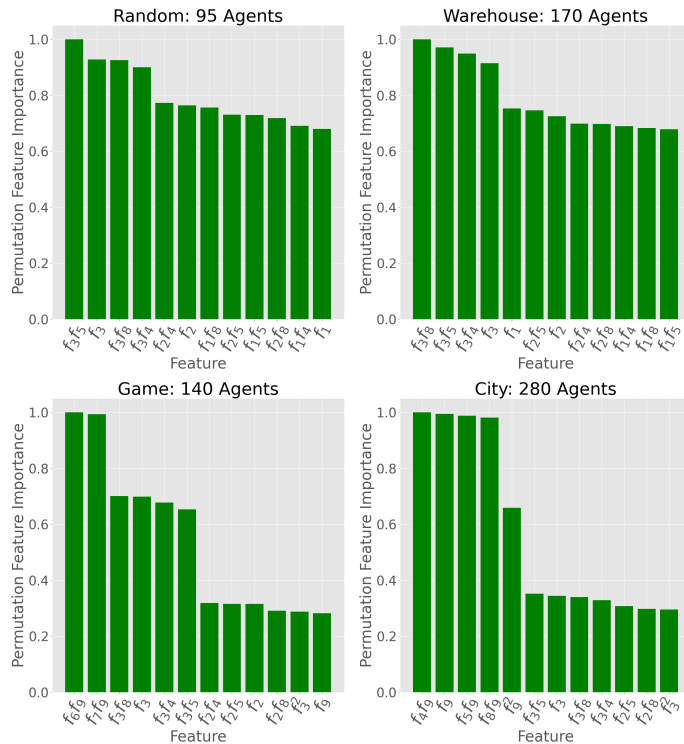
(b) Permutation feature importance of the learned ranking functions for different grid maps.

Figure 2.4: Feature importance plots. We restate the definitions of some atomic features here (see Section 2.7.1.2 for the full list): f_1 is the number of conflicts, f_2 is the number of pairs of agents that have at least one conflict with each other, f_3 is the number of agents that have at least one conflict with other agents, and f_9 is the depth of the CT node.

features of the ranking functions when varying the numbers of agents for a single grid map, as shown in Figure 2.4a. We choose the maze map as a representative example to show that the learned node-selection strategies change as the number of agents increases. For 45 agents, the most important features are related to f_1 (the number of conflicts), followed by some features related to f_2 (the number of pairs of agents that have at least one conflict with each other). For both 65 and 85 agents, the top 6 features are related to f_2, followed by some features related to f_3 (the number of agents that have at least one conflict with other agents) for 65 agents and f_1 for 85 agents. For 105 agents, the most important features are related to f_3, followed by some features related to f_1. To show that the set of important features varies across grid maps, we study the feature importance of the ranking functions for the random, warehouse, game and city maps, as shown in Figure 2.4b. The ranking functions are for the numbers of agents used in Figure 2.2. For the random and warehouse maps, the most important features are related to f_3, and the feature importance drops after the 4th feature. For the city map, the most important features include five features related to f_9 (the depth of the CT node). For the game map, the two most important features are also related to f_9, followed by some features related to f_3.

2.8   Learning to Select Agent Sets for MAPF-LNS

In this section, we introduce MAPF-ML-LNS to show how our framework can be applied to improving selecting agent sets in MAPF-LNS. We have introduced how our framework can be applied to improve optimal and bounded-suboptimal MAPF search algorithms. However, both CBS and ECBS often run too slowly due to proving (sub)optimality during the search, especially when solving large MAPF instances with high agent or obstacle densities. To tackle these issues, researchers have studied anytime unbounded-suboptimal MAPF search algorithms. The appeal of an anytime MAPF search algorithm is that it first finds an initial solution quickly using any existing MAPF search algorithm and, if more runtime is available,

then improves the solution quality over time. MAPF-LNS [117] is a state-of-the-art anytime MAPF search algorithm that uses Large Neighborhood Search (LNS).

MAPF-LNS uses an agent-based and a map-based heuristic to select agent sets to destroy. The number of agent sets that could be generated by these (randomized) agent-set selection strategies can be exponential in the cardinality of the agent sets, and MAPF-LNS randomly selects one of them (namely the one that is first randomly generated). However, some agent sets might not improve the solution as much as other agent sets and even result in no improvement at all, even if they are all generated by the same agent-set selection strategy. We apply the framework introduced in Section 2.5 and tailor it for agent set selection in MAPF-LNS. We then empirically demonstrate the effectiveness and efficiency of MAPF-ML-LNS.

2.8.1 Machine Learning Methodology

Our goal is to learn an agent-set selection strategy to improve the solution faster than the existing ones. The agent-set selection strategy is applied in every iteration of MAPF-LNS. Thus, we represent the state of the search with the incumbent solution $s = P$ (the solution with the lowest sum of costs found so far in the search), and we let the set of actions $A(P) = \mathcal{B}(P)$ be the sets of agent sets that can be selected by the strategy. The size of $\mathcal{B}(P)$ is exponential in the cardinality of the agent sets. We first propose a sampling-based expert for agent-set selection. The expert reduces the size of $A(P)$ by down-sampling a collection of agent sets using one of the two agent-set selection strategies in MAPF-LNS. It then replans the paths of all agents in the sampled agent sets and selects the agent set that reduces the sum of costs the most. However, the expert is time-consuming to compute. We therefore learn to imitate the expert with a linear ranking function. Finally, we use the learned ranking function to guide agent-set selection during the search.
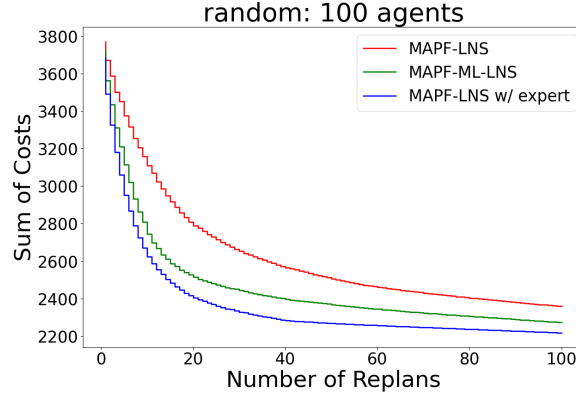
Figure 2.5: Evolution of the solution quality as a function of the number of replans for MAPF-LNS, MAPF-ML-LNS and MAPF-LNS with the expert.

2.8.1.1 Expert for Agent-Set Selection

Given a MAPF instance and its incumbent solution P, the expert for agent-set selection first calls the agent-set selection strategies to sample a collection of S agent sets $\mathcal{B}(P)$, where S is a constant that is set to 20 throughout the experiments. Each agent-set sample is generated by a randomized agent-set selection strategy chosen from the agent-based and map-based heuristics[¶] with uniform probability, and its size is chosen uniformly at random from 5 to 16. For each of the S agent sets, the expert replans the paths of the agents in it and records the cost improvement, i.e., the resulting decrease in the sum of costs. Finally, the expert outputs the agent set with the highest rank, i.e., the one with the largest cost improvement.

We replace the agent-set selection in MAPF-LNS (Lines 5-6 in Algorithm 2) with the expert and compare the resulting version of MAPF-LNS with the expert against MAPF-LNS for 100 agents on the random map "random-32-32-10", which is a 32×32 grid map from the MAPF benchmark set [181] with 10% randomly blocked cells. The grid map is shown in Table 2.9. We follow the experimental setup introduced in Section 2.8.2. We allocate a budget of 100 replans to each algorithm (instead of a runtime limit). Figure 2.5 shows how the average sum of costs changes after each replan. The average runtime of MAPF-LNS is

---

[¶]We started this work when an earlier version [118] of MAPF-LNS came out that uses only the two heuristics. MAPF-LNS [117] actually uses a third heuristic that randomly generates agent sets. We tried adding this heuristic to the expert but saw little improvement in the results.

Feature Descriptions	Count
Static Features	6
Distance between a_i's start and goal vertices.	1
Row and column numbers of a_i's start and goal vertices.	4
Degree of a_i's goal vertex.	1
Dynamic Features	10
Delay of a_i.	1
Ratio between the delay of a_i and the distance between a_i's start and goal vertices.	1
Minimum, maximum, sum and average of the heat values of the vertices on a_i's path p_i: The heat value of vertex $v \in V$ is the number of time steps that v is occupied by an agent. The heat value of a vertex counts multiple times in the sum and average if the vertex is visited by the agent multiple times until it no longer leaves the goal vertex.	4
Number of time steps that a_i is on a vertex with degree j ($1 \leq j \leq 4$) until it no longer leaves the goal vertex.	4

Table 2.7: Agent a_i's features with respect to instance I and incumbent solution $P_I = \{p_i : i \in [k]\}$. The counts are the numbers of features contributed by the corresponding entries.

0.8 seconds, while the one of MAPF-LNS with the expert is more than 16 seconds, which is too slow to be useful for MAPF solving. However, the huge difference between the curves of MAPF-LNS (red) and MAPF-LNS with the expert (blue) in Figure 2.5 suggests that, if we could learn an ML model that approximates the expert accurately with a small computational overhead during MAPF solving, then a version of MAPF-LNS with ML-guided LNS might be able to improve the solution quality faster early in the search than MAPF-LNS. The curves of MAPF-ML-LNS (green) in Figures 2.5 and 2.6 show that this is indeed possible.

2.8.1.2 Data Collection

Given an instance I, the incumbent solution P and the number S of agent sets to sample, we describe the subroutine collectData(I, P, S) that will be used to collect features and labels for P in our learning algorithm.

For incumbent solution P, we sample a collection $\mathcal{B}(P)$ of S agent sets using the expert. For each $B \in \mathcal{B}(P)$, we compute a feature vector $\phi_P(B)$ and a ground-truth label $y_P(B)$ transformed from the expert's ranking.

Features   To compute the feature vector $\phi_P(B)$ of a given agent set $B \in \mathcal{B}(P)$, we first compute a set of 16 agent features for each agent $a_i \in \{a_1, \dots, a_k\}$, which are summarized in Table 2.7. We then divide the set of agents into two subsets, B and $\{a_1, \dots, a_k\} \setminus B$. For each subset, we compute the minimum, maximum, sum and average of the value of each of the 16 agent features over all agents in the subset, resulting in $4 \times 16 = 64$ features for the subset and $p = 2 \times 64 = 128$ features for both subsets. We perform a linear transformation to normalize the value of each feature to the range of $[0, 1]$ across all agent sets in $\mathcal{B}(P)$, where the minimum value of that feature gets transformed into a 0 and the maximum value gets transformed into a 1. We then concatenate them to obtain the feature vector $\phi_P(B)$.

Labels   A ground-truth label $y_P(B)$ is a value assigned to each agent set $B \in \mathcal{B}(P)$, such that agent sets that result in higher cost improvements have smaller values. We use a simple and intuitive soft labeling scheme following previous work [102]: Let α and β ($\alpha \geq \beta$) be the cost improvements of the agent sets ranked at the 75 and 50 percentiles by the expert, respectively, and set $y_P(B) = \mathbf{1}_{[\Delta_B \geq \alpha]} + \mathbf{1}_{[\Delta_B \geq \beta]}$, where Δ_B is the cost improvement of B (in our study, we achieved similar results when labeling with 75, 50 and 25 percentiles as well as 80 and 50 percentiles). This labeling scheme assigns label 2 to the agent sets ranked in the top 25%, label 1 to the ones ranked in the top 50% but not the top 25% (i.e., the ones better than a choice at random) and label 0 to the rest. Our labeling scheme relaxes the definition of the best agent set and allows us to learn a ranking function that focuses on selecting only high-ranking agent sets with respect to their cost improvements and avoids having to correctly rank agent sets with small or no cost improvements. We tried using binary labels, e.g., $y_P(B) = \mathbf{1}_{[\Delta_B \geq \alpha]}$, and using the cost improvements Δ_B as the labels but did not get as good performance as the one we proposed.

2.8.1.3   Model Learning

We use imitation learning to learn a strategy for agent-set selection. We adapt the data-aggregation algorithm [168] combined with the forward training algorithm [167] to our use case.

Algorithm 5 Training Algorithm

1: **Input:** Training instance set $\mathcal{I}_{\mathsf{Train}}$, number R of iterations and number S of agent set samples
2: **for** $I \in \mathcal{I}_{\mathsf{Train}}$ **do**
3:      $P \leftarrow \mathsf{runInitialSolver}(I)$
4:      Record P as the incumbent solution of I
5: $D = \emptyset$
6: **for** $r = 1$ to R **do**
7:      **for** $I \in \mathcal{I}_{\mathsf{Train}}$ **do**
8:          $P \leftarrow$ incumbent solution of I
9:          $\mathsf{collectData}(I, P, S)$ $\triangleright$ Sample S agent sets for instance I and collect their features and labels using the expert.
10:          $D \leftarrow D \cup \{P\}$ $\triangleright$ Then, add the state P to the training dataset.
11:      Train $\pi^{(r)}$ with D
12:      **for** $I \in \mathcal{I}_{\mathsf{Train}}$ **do**
13:          $P \leftarrow$ incumbent solution of I
14:          $\mathcal{B}(P) \leftarrow \emptyset$
15:          **for** $i = 1$ to S **do**
16:              $\mathcal{H} \leftarrow$ uniformly select one of the two heuristics
17:              $\mathcal{B}(P) \leftarrow \mathcal{B}(P) \cup \mathsf{selectAgentSet}(I, \mathcal{H})$
18:          $B \leftarrow \arg\max_{B' \in \mathcal{B}(P)} \pi^{(r)}(\phi_P(B'))$
19:          $P^- \leftarrow \{p_i \in P : a_i \in B\}$
20:          $P^+ \leftarrow \mathsf{runReplanSolver}(I, B, P \setminus P^-)$
21:          **if** $\sum_{p \in P^+} l(p) < \sum_{p \in P^-} l(p)$ **then**
22:              $P \leftarrow (P \setminus P^-) \cup P^+$
23:          Update P as the incumbent solution of I
24: $\pi \leftarrow \mathsf{validate}(\{\pi^{(1)}, \ldots, \pi^{(R)}\})$
25: **return** π

The training algorithm, shown in Algorithm 5, takes as input a set $\mathcal{I}_{\mathsf{Train}}$ of training instances and

runs for R iterations. We fix the grid map and the number of agents for the training instances, where

the start and goal vertices of the agents are drawn i.i.d. from a given distribution. The training algorithm

first computes an initial solution P for each $I \in \mathcal{I}_{\mathsf{Train}}$ (Lines 2-4). In each iteration r ($1 \leq r \leq R$), it

collects training data for each $I \in \mathcal{I}_{\mathsf{Train}}$ by probing the expert and recording its decision with respect to

the incumbent solution P of I as well as the features of the agent sets sampled by the expert (Lines 7-10).

Then, it trains a ranking function $\pi^{(r)}$ that minimizes a loss function over the aggregated training data

D (Line 11). To improve P, it evaluates all agent sets $\mathcal{B}(P)$ to select an agent set B using $\pi^{(r)}$ (Line 18),

replans the paths of all agents in B (Line 16) and updates P if the solution improves (Lines 21-23). After

R iterations, it returns the ranking function that performs best during validation (Lines 24-25). Algorithm

Algorithm 6 MAPF-ML-LNS

1: **Input:** MAPF instance I, ranking function π and number S of agent set samples
2: $P = \{p_i : i \in [k]\} \leftarrow$ runInitialSolver(I)
3: Initialize the weights ω of the agent-set selection strategies
4: **while** runtime limit not exceeded **do**
5: $\quad \mathcal{B}(P) \leftarrow \emptyset$
6: $\quad$ **for** $i = 1$ to S **do**
7: $\quad\quad \mathcal{H} \leftarrow$ selectDestroyHeuristic(w)
8: $\quad\quad \mathcal{B}(P) \leftarrow \mathcal{B}(P) \cup$ selectAgentSet$(I, \mathcal{H})$
9: $\quad$ Compute $\pi(\phi_P(B))$ for all $B \in \mathcal{B}(P)$
10: $\quad$ **for** $B \in \mathcal{B}(P)$ in descending order of $\pi(\phi_P(B))$ **do**
11: $\quad\quad P^- \leftarrow \{p_i : a_i \in B\}$
12: $\quad\quad P^+ \leftarrow$ runReplanSolver$(I, B, P \setminus P^-)$
13: $\quad\quad$ Update the weights ω of the agent-set selection strategies
14: $\quad\quad$ **if** $\sum_{p \in P^+} l(p) < \sum_{p \in P^-} l(p)$ **then**
15: $\quad\quad\quad P \leftarrow (P \setminus P^-) \cup P^+$
16: $\quad\quad\quad$ **break**
17: **return** P

5 repeatedly determines a ranking function that makes good decisions in those situations encountered in previous iterations when using the previously learned ranking functions to guide agent-set selection.

Given the dataset D collected during training, we follow the formulation in Section 2.5 to learn a linear ranking function

$$\pi : \mathbb{R}^p \to \mathbb{R} : \pi(\phi_P(B)) = \boldsymbol{w}^\mathsf{T} \phi_P(B)$$

with parameter $\boldsymbol{w} \in \mathbb{R}^p$, that minimizes the loss function

$$L(\boldsymbol{w}) = \sum_{P \in D} l(y_P, \hat{y}_P) + \frac{C}{2} ||\boldsymbol{w}||_2^2.$$

To compute $l(y_P, \hat{y}_P)$, we consider the set of pairs $\mathcal{P}_P = \{(B', B'') : B', B'' \in \mathcal{B}(P) \wedge y_P(B') > y_P(B'')\}$ and calculates it as the fraction of swapped pairs

$$l(y_P, \hat{y}_P) = \frac{|\{(B', B'') \in \mathcal{P}_P : \hat{y}_{B'} \leq \hat{y}_{B''}\}|}{|\mathcal{P}_P|}.$$

2.8.1.4 ML-Guided Search

After learning the ranking function π, we deploy it in MAPF-ML-LNS. MAPF-ML-LNS is summarized in Algorithm 6. In each iteration, given the incumbent solution P, MAPF-ML-LNS samples a collection $\mathcal{B}(P)$ of S agent sets using the two agent-set selection strategies (Lines 6-8) and computes the predicted score $\pi(\phi_P(B))$ for each agent set $B \in \mathcal{B}(P)$ (Line 9). The agent-set selection strategies are chosen from the agent-based and map-based heuristics with probabilities according to the weights ω maintained by adaptive LNS. MAPF-ML-LNS replans the paths for the agents in agent sets (Line 12) in descending order of the predicted scores of the agent sets. If a new incumbent solution is found, it discards the remaining agent sets, recomputes the agent features and continues to the next iteration (Lines 14-16).

Given an instance I and its incumbent solution $P = \{p_i : i \in [k]\}$, the time complexity of computing the 16 agent features is bounded by $O(k + \sum_{i \in [k]} l(p_i))$, which is linear in the number of agents and the sum of costs. The agent features need to be recomputed only if a new incumbent solution is found. For each $B \in \mathcal{B}(P)$, $\phi_P(B)$ can be computed in time $O(|B|p)$.This could be done by pre-processing the largest 16 values, the smallest 16 values and the sum of feature values for each agent feature when computing them.

Discussion One of the main contributions in this section is the expert for agent-set selection. Previous works have applied imitation learning to improve LNS for MILP solving [179, 177]. For MILP, we will see in Chapter 3 that there is an existing expert called Local Branching [56] to guide selecting which subset of variables to reoptimize next and [179] learns to predict the subset given by Local Branching. However, for MAPF, there is no existing expert. Therefore, we design one that leverages spatio-temporal information by using two heuristics and frame our ML problem as learning an agent-set selection strategy to guide destroying a part of the solution in LNS. Subsequently, this allows us to use a lightweight linear ML model, such as SVM$^{\mathsf{rank}}$, that is easy to train and fast to evaluate during MAPF solving. We do not learn how to construct agent sets or predict the cost improvement of given agent sets since these are much

more complicated ML problems that require using larger ML models, such as deep neural networks. We experimented with graph convolutional networks for these tasks on an agent dependency graph [120] and ended up with good ML performance but an undesirably large computational overhead due to their high model complexity, rendering them useless without further in-depth engineering.

2.8.2 Empirical Evaluation

In this subsection, we demonstrate the efficiency and effectiveness of MAPF-ML-LNS through experiments. In the following, we introduce our evaluation setup and then present the results.

2.8.2.1 Setup

We implement MAPF-ML-LNS in C++ and conduct our experiments on a 2.4 GHz Intel Core i7 CPU with 16 GB RAM. We compare against MAPF-LNS on five grid maps of different sizes and structures from the MAPF benchmark set [181]: (1) the random map "random-32-32-10"; (2) the game map "den520d", which is a 257×256 grid map from the video game *Dragon Age: Origins*; (3) the city map "Paris_1_256", which is a 256×256 grid map of Paris; (4) the game map "ost003d", which is a 194×194 grid map from the video game *Dragon Age: Origins*; and (5) the warehouse map "warehouse-10-20-10-2-1", which is a 163×63 grid map with 200 10×2 rectangular obstacles. The five grid maps are shown in Table 2.9. Compared to the MAPF instances used in Section 2.7.2, we use grid maps of similar sizes and obstacle densities but increase the agent densities since MAPF-LNS scales better than ECBS.

MAPF-LNS and MAPF-ML-LNS use the same setup. For the initial search algorithms and each grid map, we follow [117] and select the MAPF search algorithm from PP, PPS and EECBS that has the highest success rate on the instances with the largest number of agents within a runtime limit of 10 seconds as reported by them. We use that MAPF search algorithm consistently for training and MAPF solving. That is, we use PP as the initial search algorithm for the city and both game maps (den520d and ost003d), and

Grid Map	Training k	Average Ranking	Improving Choice	Regret
random	100	6.5/20	90%	25%
den520d	200	7.0/20	96%	33%
city	250	6.7/20	99%	19%
ost003d	100	5.4/20	91%	26%
warehouse	100	6.0/20	90%	28%

Table 2.8: Validation results for the learned ranking function π. "Training k" is the number of agents of the training instances. "Average ranking" is the average rank of the first agent set selected by π among the $S = 20$ agent sets. "Improving choice" is the fraction of times π selects an agent set that results in a positive cost improvement. "Regret" is calculated as the average of 100% minus the cost improvement achieved by π as a percentage of the cost improvement achieved by the expert.

PPS for the random and warehouse maps. We use PP as the replan search algorithm for all grid maps, since PP dominates the other MAPF search algorithms, namely CBS and EECBS [117].

During training, we run Algorithm 5 for $R = 100$ iterations. For each grid map, we use $|\mathcal{I}_{\mathsf{Train}}| = 16$ instances with a fixed number of agents. The number of agents k of the training instances is reported in Table 2.8. Since we use a randomized version of PP that uses random agent priorities, the cost improvement of each agent set used for creating its label is the average taken over 6 runs. We use regularization parameter $C = 0.1$ and the default values for the other parameters in the SVM$^{\mathsf{rank}}$ solver. We also tried $C \in \{0.01, 0.001\}$ and achieved similar results. It takes 2 to 8 hours, depending on the grid map, to run Algorithm 5 on a single CPU. If collecting training data for the 16 instances were done in parallel on 16 CPUs in each iteration (Line 7 in Algorithm 5), the training time could be reduced to less than 1 hour.

During validation, we evaluate $\pi_1, \ldots, \pi_T$ on the validation data and return the ranking function π that selects agent sets with the highest average ranking. We run MAPF-LNS with the expert for 100 iterations on 4 MAPF instances from the same distribution as the training instances. The validation results for π are summarized in Table 2.8. During testing, we use 25 MAPF instances and set a runtime limit of 60 seconds per instance. For both MAPF-LNS and MAPF-ML-LNS, the runtime limit for finding the initial solution is set to 10 seconds. Those instances for which they fail to find an initial solution within 10 seconds are considered unsolvable and not included in our results. We use the same random seed to ensure that both

methods compute the same instances and initial solutions. The runtime limit of PP per replan is set to 2 seconds for the warehouse map and 0.6 seconds for the other grid maps initially and then adaptively set to twice the average runtime of all successful replans so far after the first 30 successful replans. We use the adaptive runtime limits for replanning since we observe that the runtime for unsuccessful replans is longer than that for successful replans, and the runtime for replans is different on different maps. During both training and MAPF solving, when generating an agent set using the agent-set selection strategies in LNS, we draw its cardinality uniformly from 5 to 16. We sample $S = 20$ agent sets in each iteration of MAPF-ML-LNS.

2.8.2.2 Results

Our results provide answers to the following questions:

1. If the grid map is known in advance, can we learn a ranking function that performs well on the same grid map with the same and different numbers of agents?

2. If the grid map is unknown in advance, can we learn a ranking function from other grid maps that performs well on the unknown one?

We therefore learn two ranking functions with SVM$^\text{rank}$ for each grid map, namely a ranking function trained on MAPF instances on that grid map (resulting in MAPF search algorithm ML-S) and a ranking function trained on MAPF instances from the other four grid maps (resulting in MAPF search algorithm ML-O).

Solution Quality and the Speed of Improving the Solution An important metric for evaluating the performance of an anytime MAPF search algorithm is its speed of improving the solution. Let $\mathcal{I}_\text{Test}$ be the set of test instances and, for each $I \in \mathcal{I}_\text{Test}$, let $t_{I,\text{init}}^{\mathcal{S}}$, $SOC_I^{\mathcal{S}}(t)$ and $SOD_I^{\mathcal{S}}(t)$ be the runtime needed to find the initial solution, the sum of costs and the sum of delays of the solution at runtime t, respectively,

Grid Map	k	AUC Ratio		Win/Loss		Sum of Agents' Delay (Suboptimality)		
		ML-S	ML-O	ML-S	ML-O	MAPF-LNS	ML-S	ML-O
random	**100**	**1.15±0.23**	**1.12±0.20**	**20/5**	**20/5**	30 (1.01)	**28 (1.01)**	**28 (1.01)**
	150	1.14±0.12	1.07±0.12	22/3	21/4	105 (1.03)	**96 (1.03)**	**96 (1.03)**
	200	1.03±0.10	1.07±0.19	15/9	15/9	309 (1.07)	**275 (1.06)**	**270 (1.06)**
	250	0.98±0.17	0.95±0.12	10/15	8/17	806 (1.15)	843 (1.15)	845 (1.15)
	300	1.13±0.14	1.06±0.15	18/6	13/11	4,460 (1.67)	**3,754 (1.56)**	4,301 (1.61)
	350	0.99±0.08	0.94±0.08	11/12	6/17	21,310 (3.78)	22,234 (3.90)	23,674 (4.08)
den520d	**200**	**1.97±0.56**	**1.75±0.53**	**23/2**	**24/1**	64 (1.00)	65 (1.00)	66 (1.00)
	300	1.62±0.55	1.45±0.43	21/4	20/5	400 (1.01)	**298 (1.00)**	**328 (1.01)**
	400	1.65±0.54	1.31±0.30	25/0	22/3	1,327 (1.02)	**778 (1.01)**	**1,121 (1.01)**
	500	1.25±0.35	1.13±0.22	19/6	18/7	3,616 (1.04)	**2,676 (1.03)**	**3,281 (1.03)**
	600	1.10±0.15	1.10±0.08	18/7	24/1	8,134 (1.08)	**6,654 (1.06)**	**6,967 (1.07)**
	700	1.07±0.06	1.05±0.06	22/3	20/5	12,558 (1.10)	**11,785 (1.10)**	**11,535 (1.09)**
city	**250**	**1.75±0.41**	**1.14±0.32**	**22/3**	**14/11**	229 (1.00)	**110 (1.00)**	**128 (1.00)**
	350	1.12±0.34	1.02±0.24	19/6	14/11	469 (1.01)	**372 (1.01)**	**368 (1.01)**
	450	1.30±0.35	1.01±0.22	19/6	13/12	763 (1.01)	**629 (1.01)**	**753 (1.01)**
	550	1.05±0.18	1.06±0.24	16/9	14/11	1,932 (1.02)	2,056 (1.02)	**1,536 (1.01)**
	650	1.08±0.13	1.10±0.25	17/8	17/8	3,274 (1.03)	**3,041 (1.02)**	**3,033 (1.02)**
	750	1.07±0.14	1.09±0.08	17/6	19/4	8,371 (1.06)	**8,363 (1.06)**	**7,413 (1.05)**
ost003d	**100**	**1.28±0.33**	**1.17±0.28**	**21/4**	**15/10**	42 (1.00)	42 (1.00)	42 (1.00)
	200	1.43±0.36	1.20±0.27	19/4	17/6	458 (1.01)	**332 (1.01)**	**372 (1.01)**
	300	1.14±0.19	1.16±0.16	16/8	20/4	2,509 (1.05)	**2,379 (1.05)**	**2,152 (1.04)**
	400	1.05±0.08	1.06±0.08	17/6	17/6	6,907 (1.11)	**6,584 (1.10)**	**6,417 (1.10)**
	500	1.02±0.03	1.04±0.05	15/7	16/6	14,750 (1.19)	**14,431 (1.19)**	**14,251 (1.18)**
	600	1.02±0.03	1.03±0.04	14/6	16/4	24,684 (1.27)	**24,468 (1.27)**	**24,401 (1.27)**
warehouse	**100**	**1.35±0.33**	**1.25±0.30**	**20/5**	**20/5**	57 (1.01)	**37 (1.00)**	**37 (1.00)**
	150	1.21±0.24	1.14±0.22	18/7	16/9	295 (1.02)	**195 (1.01)**	**217 (1.02)**
	200	1.19±0.22	1.05±0.13	21/4	15/10	925 (1.06)	**736 (1.05)**	**842 (1.05)**
	250	1.17±0.20	1.11±0.18	17/8	16/9	1,817 (1.09)	**1,595 (1.08)**	**1,805 (1.09)**
	300	1.18±0.21	1.13±0.19	17/8	18/7	4,719 (1.20)	**3,852 (1.16)**	**3,547 (1.15)**
	350	1.07±0.10	1.02±0.07	15/9	13/11	12,004 (1.43)	**10,191 (1.36)**	12,143 (1.43)

Table 2.9: The average ratios of the AUCs of MAPF-LNS and variants of MAPF-ML-LNS (ML-S and ML-O) with their standard deviations, the win/loss counts with respect to the AUCs and the average sums of delays with the average suboptimalities for a runtime limit of 60 seconds. All entries take only the solved MAPF instances into account. We bold the number of agents k on which ML-S is trained and the entries where a variant of MAPF-ML-LNS outperforms MAPF-LNS.
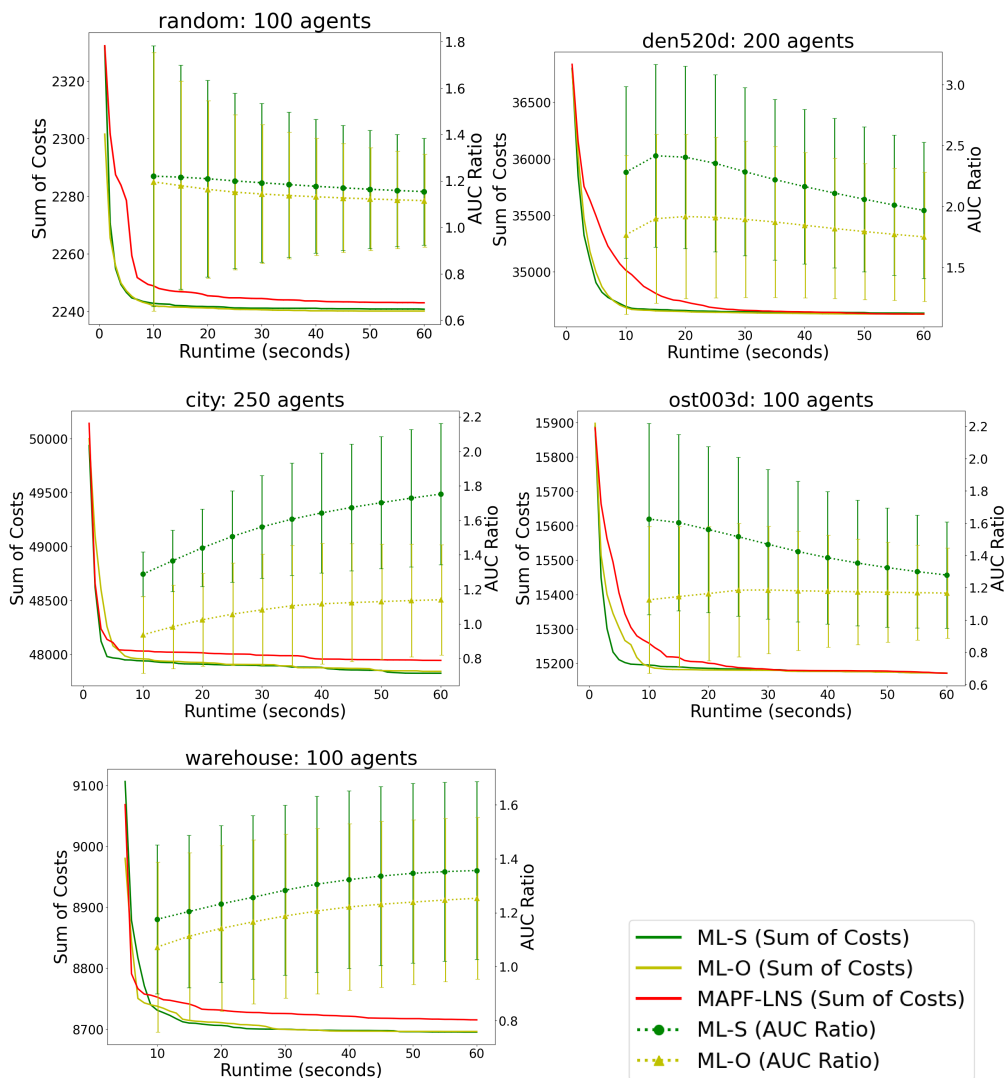
Figure 2.6: Evolutions of the sum of costs (solid curves with the y-axis on the left side, smaller is better) from 1 second to 60 seconds for MAPF-LNS, ML-S and ML-O, averaged over all solved instances, and the average ratio of the AUCs of MAPF-LNS and one of ML-S and ML-O (dotted curves with the y-axis on the right side, greater than 1 is better), also averaged over all solved instances, as a function of the runtime. The error bars represent the standard deviation.

when solving instance I using MAPF search algorithm $\mathcal{S}$. Following [117], we compute the Area Under the Curve (AUC) of the sum of delays as a function of the runtime of MAPF search algorithm $\mathcal{S}$ on each instance I, which is formally defined as $AUC_I^{\mathcal{S}}(t_{\mathsf{limit}}) = \int_{t_{I,\mathsf{init}}^{\mathcal{S}}}^{t_{\mathsf{limit}}} SOD_I^{\mathcal{S}}(t)\mathrm{d}t$, where t_{limit} is the runtime limit (60 seconds). The smaller the AUC, the higher the speed of improving the solution is. In Table 2.9, we report the average ratios of the AUCs of MAPF-LNS and our MAPF search algorithms, the win/loss counts with respect to the AUC and the average sums of delays with the average suboptimalities over all solved test instances[||]. The win/loss counts are the numbers of instances where the AUCs of ML-S or ML-O are smaller/larger than those of MAPF-LNS. The suboptimalities are overestimated values calculated as the ratio between the final sum of costs and the sum of distances between the agents' start and goal vertices. On the city and both game maps (den520d and ost003d), the AUCs of MAPF-LNS are 43% to 97% worse than the ones of ML-S. On these three maps, ML-S substantially outperforms MAPF-LNS also with respect to the win/loss counts and, for almost all tested numbers of agents, with respect to the final solution qualities. On the random and warehouse maps, ML-S outperforms MAPF-LNS with respect to all metrics, except for a few cases with large numbers of agents (250 and 350 agents on the random map). Even though ML-S learns the ranking functions on MAPF instances with a fixed number of agents, they generalize well to MAPF instances with larger numbers of agents on the same grid map and outperform MAPF-LNS in almost all cases. ML-O also substantially outperforms MAPF-LNS. ML-O, without seeing the test grid map during training, is competitive with ML-S and even outperforms it sometimes on both game maps and the city map. For the random map, the improvement of MAPF-ML-LNS over MAPF-LNS is not as substantial as for the other grid maps, especially on MAPF instances with large numbers of agents. We tried retraining the ranking functions on MAPF instances with larger numbers of agents (e.g., 250 agents for the random map) but achieved similar results. It is future work to improve the effectiveness of MAPF-ML-LNS on this grid map.

[||]All search algorithms have the same set of solved instances since they use the same initial search algorithm with the same random seeds.

To demonstrate the effectiveness of our MAPF search algorithms further, we show the average sum of costs for MAPF-LNS, ML-S and ML-O in Figure 2.6 together with the average ratios between the AUCs of MAPF-LNS and one of ML-S and ML-O as functions of the runtime limit t_{limit}, i.e., $\frac{1}{|\mathcal{I}_{\text{Test}}|} \sum_{I \in \mathcal{I}_{\text{Test}}} SOC_I^{\mathcal{S}}(t_{\text{limit}})$ and $\frac{1}{|\mathcal{I}_{\text{Test}}|} \sum_{I \in \mathcal{I}_{\text{Test}}} \frac{AUC_I^{\text{MAPF-LNS}}(t_{\text{limit}})}{AUC_I^{\mathcal{S}}(t_{\text{limit}})}$ for each $\mathcal{S} \in \{\text{ML-S, ML-O}\}$. In these cases, ML-S and ML-O establish advantages early in the search and substantially outperform MAPF-LNS for several shorter runtime limits, e.g., 20 or 30 seconds.

Grid Map	k	Number of Replans	
		MAPF-LNS	ML-S
random	100	19,075	15,892
	200	6,398	5,673
	300	1,002	711
den520d	200	1,138	932
	400	633	620
	600	401	374
city	250	1,978	1,452
	450	1,314	1,101
	650	794	783
ost003d	100	1,398	1,044
	300	419	317
	500	168	138
warehouse	100	3,152	2,706
	200	874	695
	300	241	188

Table 2.10: The average number of replans of MAPF-LNS and ML-S for a runtime limit of 60 seconds.

The runtime overhead of MAPF-ML-LNS induced by computing the features and evaluating the ranking function is small. Table 2.10 shows the average number of replans of MAPF-LNS and ML-S for a runtime limit of 60 seconds. MAPF-ML-LNS performs fewer replans than MAPF-LNS on average. These results suggest that our learned ranking functions select agent sets more effectively since they improve the solutions faster and achieve better solution qualities than MAPF-LNS with fewer replans.

Feature Importance   Finally, we study the feature importance of the learned ranking function for ML-S for each grid map, measured by the absolute values of the learned feature weights. It makes sense to do

so since the features are normalized. Features related to the delays are the most important ones for all five grid maps. The other important features are related to the costs of the paths, the ratios between the delays and costs, the sums of the heat values on the paths and the numbers of time steps that the agents are on a vertex with degree 2 or 3 (see Table 2.7 for definitions).

2.9   Learning to Prioritize Agents for PP

In this section, we introduce PP+ML to show how our framework can be applied to improve the assignment of priorities to agents in prioritized planning (PP) [175]. PP is one of the fastest algorithms for solving MAPF suboptimally. It is based on a simple planning scheme [49]: It assigns each agent a unique priority and computes, in descending priority ordering, each agent's cost-minimal path that avoids conflicts with both static obstacles and the already-planned agents (moving obstacles). Because of its computational efficiency and simplicity, PP remains the most commonly-adopted MAPF algorithm in practice [197]. For example, PP is commonly used to find the initial solution in LNS-based MAPF search algorithms [117, 91]. However, its solution quality is sensitive to the predetermined priority ordering. Good priority orderings can yield (near-)optimal solutions, whereas bad priority orderings can lead to solutions with large sums of costs or even failures to find any solution for solvable MAPF instances, as shown in Figure 2.7.

Existing PP algorithms use either randomized assignments or greedy heuristics to determine the priority ordering, such as the query-distance heuristic [17], least-option heuristic [191, 196] and start-and-goal-conflict heuristic [24, 116]. However, these hand-crafted heuristics have been developed in the context of specific usage scenarios, and none of them dominates the others in all cases in terms of the success rate and solution quality (measured by the sum of costs). We apply our framework introduced in Section 2.5 to this task and tailor it for priority assignments for agents in PP. We then empirically demonstrate the effectiveness and efficiency of PP+ML.
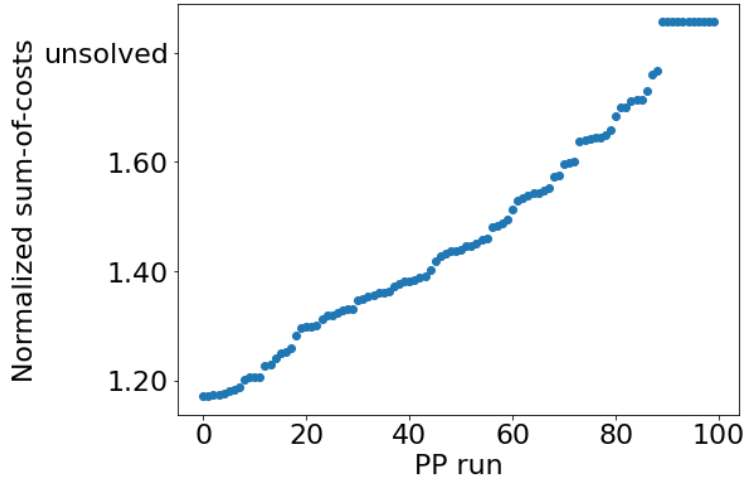
Figure 2.7: Normalized sum of costs (i.e., we normalize them by taking the ratio of the sum of costs of the solution over the sum of the lengths of the individually cost-minimal paths of all agents) of 100 PP runs with different random priority orderings on MAPF instance "room-32-32-4-random-1.scen" from [181] with 20 agents, sorted in increasing order of their normalized sums of costs. PP runs that fail to find a solution are shown on the top of the plot.

2.9.1 Machine Learning Methodology

Our goal is to learn a priority-assignment strategy that increases the success rate of PP and its solution quality compared to a human-designed strategy. In general, a priority-assignment strategy takes the MAPF instance and the already-planned paths as input and outputs the agent that will be planned next (i.e., it assigns the highest priority to this agent among the remaining ones). The state of the search is represented by the MAPF instance and the already-planned paths. The actions are the agents that have not been planned yet. For example, the least-option heuristic recalculates the number of path options for each agent every time a new path is planned [191]. On the other hand, the total priority ordering does not necessarily have to be either determined online during the planning process or based on the agents that have already been assigned priorities. Many previous works simplify the priority-assignment strategy to consider only the MAPF instance and, therefore, determine a total priority ordering, i.e., select all actions, before starting to plan the first cost-minimal path. Such a strategy is simple and easy to implement, and does not require extra overhead for computing the next agent to plan a path for during the planning process, in contrast to

an online strategy. In the following, we learn such a strategy. Thus, we represent the state of the search with the MAPF instance $s = I$ and let the set of actions $A(I) = \{a_1, \ldots, a_k\}$ be the set of agents. The learned priority-assignment strategy, represented by a ranking function, sequentially selects an agent with the highest rank without replacement to produce a total priority ordering in descending order. We first propose a sampling-based expert for assigning agents' priorities. The expert randomly samples a set of sequences of k agents without replacement to form a set of total priority orderings. It then uses PP to plan the paths for all agents for each total priority ordering and outputs a total or partial priority ordering based on the resulting sums of costs. However, the expert is time-consuming to compute. We, therefore, learn to imitate the expert with a linear ranking function. Finally, we use the learned ranking function to determine assigning agents' priorities in PP.

2.9.1.1 Expert for Assigning Agents' Priorities

Given a MAPF instance I, the expert for assigning agents' priorities outputs a priority ordering $\prec_I$ instead of a single decision, from which a sequence of decisions to be made at state I can be derived. To obtain the priority ordering $\prec_I$, the expert first runs PP repeatedly for x times with randomly generated total priority orderings on the agents in instance I. There are two variants of the expert to construct $\prec_I$. One outputs a total priority ordering and the other one outputs a partial priority ordering. We denote the two variants by O_T and O_P.

O_T sets $\prec_I$ to the total priority ordering that generates the solution with the smallest sum of costs among the x runs. It is simple and straightforward but has two drawbacks. First, the total priority ordering may be arbitrary in places. For example, if agents a_i and a_j are located far away from each other and do not collide with each other, then it does not matter which agent has the higher priority. Second, the total priority ordering is based on a single example, which may not be sufficiently robust.

Motivated by these drawbacks, we propose O_p to collect the $x' \geq 1$ samples that result in the smallest sums of costs from the x runs and generate a partial priority ordering by imposing an ordering on two agents only if swapping their priorities can decrease the sum of costs substantially. It works as follows:

For each PP run $p = 1, ..., x$, it starts with an empty partial priority ordering $\prec_I^p$. Each iteration of PP calls space-time A* [175] (i.e., A* that searches the space-time space, whose states are vertex-timestep pairs) to plan an individually cost-minimal path for a single agent a_i that avoids conflicts with the already-planned paths. When this A* search generates an A* node n with an f-value of f_n that moves a_i from one vertex to another, it checks if this move action leads to a conflict with an already-planned path, say, that of agent a_j, and, if so, prunes node n. The A* search records such pruned nodes, i.e., the pair (a_j, f_n). When the A* search terminates and returns a path p_i of length $l(p_i)$ for a_i, we collect the set of agents B^H in the recorded pairs whose f_n values are smaller than $l(p_i)$ and add $a_i \prec a_j$ for all $a_j \in B^H$ to $\prec_I^p$, for the following reason: If any agent in B^H had lower priority than a_i, then A* might find a path of length within $[f_n, l(p_i))$ for a_i, i.e., it might find a shorter path than the current one. In contrast, even if all agents not in B^H had a lower priority than a_i, then A* still cannot find a shorter path.

When we select the top x' samples, we collect the associated partial priority orderings $\prec_I^p$ for $p = 1, \ldots, x'$ and combine them into a joint partial priority ordering $\prec_I$. To do so, we first find all pairs of agents in each $\prec_I^p$. Specifically, we convert $\prec_I^p$ to a directed acyclic graph (DAG) H_I^p, where node i represents agent a_i and each directed edge $i \to j$ represents $a_i \prec_I^p a_j$. We run the Floyd-Warshall algorithm on H_I^p to find all connected agent pairs. We then sort the agent pairs in descending order of their occurrences in the top x' samples ($a_i \prec a_j$ and $a_j \prec a_i$ are treated as two different agent pairs) and add them one by one to $\prec_I$ whenever possible. That is, if the agent pair is $a_i \prec a_j$, we add it to $\prec_I$ iff agents a_i and a_j are not comparable in $\prec_I$. We also record the occurrences and use $\#(a_i \prec a_j)$ to represent how often $a_i \prec a_j$ occurs in the x' priority orderings.
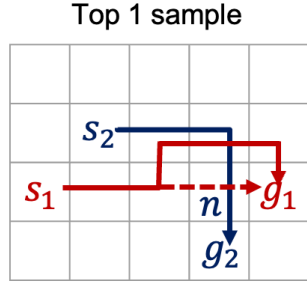
Top 1 sample

(a) Assume in the top 1 sample of the PP runs, agent a_2 is planned first and agent a_1 is planned second. The solid arrows represent the paths planned by space-time A^*. The dashed arrow shows a conflict between a_1 and a_2 if a_1 also takes its individually cost-minimal paths. n represents the node space-time A^* pruned away to avoid such a conflict.

PP Run Sample	Agent Pairs Added
Top 1 sample	$a_1 \prec a_2$
Top 2 sample	$a_1 \prec a_2, \ a_3 \prec a_4$
Top 3 sample	$a_2 \prec a_1$
Top 4 sample	$a_1 \prec a_2, \ a_1 \prec a_4$
Top 5 sample	$a_3 \prec a_4$

(b) An example of agent pairs added from the top 5 PP samples assuming $x' = 5$.

Occurrence	Agent Pairs
3	$a_1 \prec a_2$
2	$a_3 \prec a_4$
1	$a_1 \prec a_4, \quad \cancel{a_2 \prec a_1}$

(c) Occurrences of agent pairs added from the top 5 PP samples. $a_2 \prec a_1$ is not added to $\prec_I$ because it contradicts with $a_1 \prec a_2$ which has a higher occurrence.
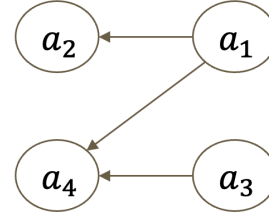
(d) The DAG representation of $\prec_I$.

Figure 2.8: An example of O_p. Assume we have a MAPF instance with $k = 4$ agents on an empty 4×5 grid map. The start and goal vertices of agents a_1 and a_2 are shown in (a).

Figure 2.8 shows an example of how O_p works. Assume we have a MAPF instance I on an empty 4×5 grid map with four agents, and we select the top $x' = 5$ samples of PP runs. The start and goal vertices of agents a_1 and a_2 are shown in Figure 2.8a. Assume that in the top 1 sample, a_2 and a_1 have the highest and second highest priorities, respectively. Therefore, a_2 takes one of its individually cost-minimal paths, as shown in Figure 2.8a (the solid blue arrow). When planning the path for a_1, the space-time A^* finds that a_1 would have a conflict with a_2 if a_1 also takes its individually cost-minimal path (the dashed red arrow). Therefore, space-time A^* prunes away the corresponding node n where $f_n = 4$. At the end, space-time A^* finds a path p_1 where $l(p_1) = 6$ (the solid red arrows). The expert then compares $l(p_1)$ with f_n. Since $l(p_1) > f_n$, it means a_1 would have found a path with a lower cost if it had not had to avoid the conflict with a_2. Therefore, the expert adds $a_1 \prec a_2$ to $\prec_I^1$. Figure 2.8b shows an example of the agent pairs added to $\prec_I^p$ from the top $p \leq 5$ samples. Figure 2.8c counts the occurrences of each agent pair in

$\prec_I^p$ ($p \leq 5$). Finally, the expert constructs the partial ordering $\prec_I$ greedily by adding agent pairs with the highest occurrences. It does not add $a_2 \prec a_1$ to $\prec_I^p$ since $a_1 \prec a_2$ is added before $a_2 \prec a_1$. Figure 2.8d shows the DAG representation of $\prec_I$.

2.9.1.2 Data Collection

The next step in our framework is to construct a training dataset from which we can learn a model that imitates the expert's output. First, we fix the graph underlying the MAPF instances that we want to solve and the number of agents. Then, we obtain a set of MAPF instances $\mathcal{I}_{\mathsf{Train}}$ for training. The training dataset $D = \mathcal{I}_{\mathsf{Train}}$ since the states of the search are represented by the instances themselves. For each $I \in D$, we run one of the experts O_T and O_P on I and derive a label $y_I(a_i)$ for each available action $a_i \in A(I)$ from the expert's priority ordering. We also compute a p-dimensional feature vector $\phi_I(a_i)$ that describes agent a_i.

Features We collect a p-dimensional feature vector $\phi_I(a_i)$ for each agent a_i and each MAPF instance $I \in \mathcal{I}_{\mathsf{Train}}$. The $p = 26$ features in our implementation are summarized in Table 2.11 and can be classified into four categories:

1. **Start-Goal Distances** Motivated by the query-distance heuristic [17] (see Section 2.3.4), we design 4 features about the graph and Manhattan distances between the start and goal vertices of a_i (Feature Group 1). We also generalize this idea to looking at the graph distances between the start/goal vertices of a_i and those of the other agents (Feature Groups 2 and 3).

2. **MDD** We consider an MDD MDD_i for agent a_i that consists of all individually cost-minimal paths from s_i to t_i, i.e., the MDD computed at the root CT node in CBS. Motivated by the least-option heuristic (see Section 2.3.4), we design 5 features about MDD_i (Feature Groups 4-6) because MDD_i captures information about the path options of a_i.

Feature Description	Count
Graph and Manhattan distances between s_i and g_i: their respective values, absolute difference and the ratio of the graph distance over the Manhattan distance.	4
Graph distance between s_i and the start vertices of the other agents: their maximum, minimum, and mean.	3
Graph distance between g_i and the goal vertices of the other agents: their maximum, minimum, and mean.	3
Sum of the widths of all levels of MDD_i.	1
Width of each level (excluding the first and the last levels) of MDD_i: their maximum, minimum, and mean.	3
Number of levels of MDD_i with width one.	1
Number of the other agents whose MDDs contain s_i.	1
Number of the other agents whose MDDs contain g_i.	1
Number of the other agents whose start vertices are in MDD_i.	1
Number of the other agents whose goal vertices are in MDD_i.	1
Number of vertex, edge and cardinal conflicts between any cost-minimal path of a_i and any cost-minimal path of one of the other agents: counted once for each agent pair and counted once for each conflict.	6
Number of vertices in MDD_i that are also in the MDD of at least one other agent.	1

Table 2.11: $p = 26$ features for agent a_i. Column "Count" reports the numbers of features contributed by the corresponding entries. We consider an MDD MDD_i for agent a_i that consists of all individually cost-minimal paths from s_i to t_i, i.e., the MDD that would have been computed at the root CT node in CBS.

3. **Start and Goal Vertices** Motivated by the start-and-goal-conflict heuristic (see Section 2.3.4), we design 4 features about the potential conflicts at the start or goal vertices of the other agents that a_i might be involved in and vice versa, namely potential conflicts between a_i and another agent a_j when a_i is at its start or goal vertex and a_j follows (one of) its individually cost-minimal paths (Feature Groups 7 and 8) or when a_j is at its start or goal vertex and a_i follows (one of) its individually cost-minimal paths (Feature Groups 9 and 10).

4. **Conflicts** We finally design 7 features about conflicts (Feature Group 11) and potential conflicts (Feature Group 12) of different types that a_i might be involved in. In particular, Feature 11 counts the number of each type of conflicts that a_i can be involved in if all agents follow their individually cost-minimal paths. Feature 12 counts the number of potential vertex conflicts that a_i can be involved in if all agents follow their individually cost-minimal paths but can wait for some time steps along

their paths. We use the numbers of these conflicts as features because they can be easily computed by reasoning about the MDDs of the agents.

We perform a linear transformation to normalize the value of each feature to the range of $[0, 1]$, where the minimum value of that feature gets transformed into a 0 and the maximum value gets transformed into a 1.

Labels   Depending on the expert used in data collection, we use different labels derived from $\prec_I$.

For expert O_T, we group the agents into $\lfloor k/m \rfloor + 1$ priority groups (where $m \in \mathbb{N}$ is a hyperparameter) by setting $y_I(a_i) = \lfloor r_i/m \rfloor$, where r_i represents that agent a_i has the r_i-th lowest priority among all agents. That is, agents with larger labels are in higher priority groups and agents with the same label are in the same priority group.

For expert O_P, we assign labels such that $y_I(a_i) > y_I(a_j)$ if $a_i \prec_I a_j$. Such assignments always exist according to Definition 2.3.1 (see Section 2.3.4) and can be found by performing a topological sort on the DAG that captures $\prec_I$.

2.9.1.3   Model Learning

Given the training dataset $D = \mathcal{I}_{\mathsf{Train}}$, we follow the formulation in Section 2.5 to learn a linear ranking function with parameters $\boldsymbol{w} \in \mathbb{R}^p$

$$\pi : \mathbb{R}^p \to \mathbb{R} : \pi(\phi_I(a_i)) = \boldsymbol{w}^\mathsf{T} \phi_I(a_i)$$

that minimizes the loss function

$$L(\boldsymbol{w}) = \sum_{I \in \mathcal{I}_{\mathsf{Train}}} l(y_I, \hat{y}_I) + \frac{C}{2}||\boldsymbol{w}||_2^2.$$

where $\hat{y}_I(a_i) = \pi(\phi_I(a_i))$ is the predicted scores for agent a_i. To compute $l(y_I, \hat{y}_I)$, we consider the set

of pairs $\mathcal{P}_I = \{(a_i, a_j) : y_I(a_i) > y_I(a_j)) \wedge a_i, a_j \in A(I)\}$. $l(y_\mathcal{T}, \hat{y}_\mathcal{T})$ is computed as follows:

$$l(y_I, \hat{y}_I) = \frac{\sum_{(a_i,a_j)\in\mathcal{P}_I : \hat{y}_I(a_i)\leq\hat{y}_I(a_j)} \tilde{w}_{a_i,a_j}}{\sum_{(a_i,a_j)\in\mathcal{P}_I} \tilde{w}_{a_i,a_j}}. \tag{2.2}$$

We train two variants of PP+ML, namely ML-T and ML-P, that are trained with data collected based on

O_T and O_P, respectively. For ML-T, we set $\tilde{w}_{a_i,a_j} = 1$ to assign uniform weights to discordant pairs in

the loss function. For ML-P, we set $\tilde{w}_{a_i,a_j} = \#(a_i \prec a_j)$ to assign weights to discordant pairs in the loss

function based on the number of occurrences of the agent pairs in $\prec_I$.

2.9.1.4 ML-Guided Search

After data collection and model learning, we apply the learned ranking function π to the feature vectors

for each test instance $I \in \mathcal{I}_{\text{Test}}$. Based on the predicted scores $\hat{y}_I : \{a_1, \ldots, a_k\} \to \mathbb{R}^k$ returned by π, we

propose two different methods to produce a total priority ordering.

Deterministic ranking. We rank the agents by their predicted scores, namely $a_i \prec a_j$ iff $\hat{y}_I(a_i) > \hat{y}_I(a_j)$.

Stochastic ranking. We use the predicted scores to produce a probability distribution and generate a

total priority ordering sequentially from agents with high priority to agents with low priority. Specifically,

we normalize the predicted scores $\hat{y}_I$ using the softmax function

$$\sigma \colon \mathbb{R}^k \to [0,1]^k : \sigma(\hat{y}_I) = \frac{(e^{\gamma\hat{y}_I(a_1)}, \ldots, e^{\gamma\hat{y}_I(a_k)})}{\sum_{j=1}^k e^{\gamma\hat{y}_I(a_j)}}, \tag{2.3}$$

where $\gamma \in \mathbb{R}^+$ is a hyperparameter. We then repeatedly assign the next highest priority to an agent

that is selected with a probability proportional to its normalized predicted score (where, of course, every

agent can be selected only once). Agents with higher normalized scores have higher probabilities of being selected earlier and thus assigned higher priority. This adds randomness to the total priority orderings and allows us to leverage the random restart scheme when experimenting with PP+ML.

Discussion   We address the limitations of PP+ML here. We will show in Section 2.9.2 that PP+ML does not outperform the baselines in some cases. First, our expert for assigning agents' priorities is based on the best of 100 random samples. Thus, there might be not enough effective samples. Especially for large grid maps, it is hard to get enough samples since evaluating each sample with the expert is slow. We thought about using the least-option heuristic as the expert since it is slow but effective. However, computing the number of path options requires high-precision computing for large grid maps which causes significant runtime overhead in the expert and makes it difficult to implement. One way to reduce such runtime overhead is to compute the number of path options modulo some large prime number (that is less than 32 bits) and repeat it with different prime numbers. We need to ensure that the number of path options is bounded by the multiplication of all prime numbers we use. We then use Chinese remainder theorem to compute the actual number of path options using high-precision computing. This method allows most arithmetic operations to be done with 64-bit integers. However, this is even more complicated to implement and we did not invest time on it. Second, the ML model does not generalize well across grid maps or different numbers of agents on the same grid map. We did not get good results when we tried curriculum learning to address this limitation similar to what we did in Section 2.7. It is also slow to collect training data on large grid maps. Therefore, for those maps, we train only one model for a fixed number of agents. Putting more effort into feature engineering and/or using a more expressive ML model than SVM^{rank} might mitigate some of these limitations.

2.9.2 Empirical Evaluation

In this subsection, we demonstrate the efficiency and effectiveness of PP+ML through experiments. In the following, we introduce our evaluation setup and then present the results.

2.9.2.1 Setup

We compare the two variants of PP+ML, namely ML-T and ML-P, against three non-ML-guided variants of PP: (1) **LH**, a query-distance heuristic where agents with longer individually cost-minimal paths have higher priority [17], breaking ties uniformly at random; (2) **SH**, a query-distance heuristic where agents with shorter individually cost-minimal paths have higher priority [137], breaking ties uniformly at random; and (3) **RND**, a heuristic that generates a random total priority ordering [16]. We implement all algorithms in C++ with the same PP code base and run experiments on Ubuntu 20.04 LTS on an Intel Xeon 8175M processor with 8 GB of memory.

We evaluate PP+ML on a set of six grid maps $\mathcal{M}$, illustrated in Table 2.13, with different sizes and structures from the MAPF benchmark [181]: (1) the random map "random-32-32-20", which is a 32×32 grid map with 20% randomly blocked cells; (2) the room map "room-32-32-4", which is a 32×32 grid map with 64 square rooms connected by single-cell doors; (3) the maze map "maze-32-32-2", which is a 32×32 grid map with two-cell-wide corridors; (4) the warehouse map "warehouse-10-20-10-2-1", which is a 161×63 grid map with 200 10×2 rectangular obstacles; (5) the first game map "lak303d" and (6) the second game map "ost003d", which are both 194×194 grid maps from the video game *Dragon Age: Origins*. We refer to the first four grid maps as small maps and to the last two as large maps.

We generate a set $\mathcal{I}_{\text{Train}}^{(M)}$ of training instances for each grid map M. For ML-T, we generate 99 training instances from each of the 25 scenarios, so $|\mathcal{I}_{\text{Train}}^{(M)}| = 2{,}475$. For ML-P, we generate one training instance from each scenario since the training loss converges already for a small training dataset, so $|\mathcal{I}_{\text{Train}}^{(M)}| = 25$.

To collect training data, we run PP $x = 100$ times, once with LH, once with SH and 98 times with RND, to solve each MAPF instance $I \in \mathcal{I}_{\text{Train}}^{(M)}$. We pick the PP run with the least sum of costs for ML-T and the top $k = 5$ PP runs with the least sums of costs for ML-P to generate the outputs of the experts. However, when we use small maps with large numbers of agents k, most of the 100 PP runs fail to find any solutions. We show in Section 2.9.2.2 that our ML models often have higher success rates than LH, SH and RND on small maps with small numbers of agents k. Therefore, when the success rate of the 100 PP runs is less than 5% for the given MAPF instances (i.e., on the random, room and maze maps with $k \geq 200$, $k \geq 125$ and $k \geq 90$, respectively), we replace 10 of the 98 RND runs with PP+ML trained on a smaller number of agents on the same map (i.e., the number of agents shown on the previous row of the row in Tables 2.12 and 2.13 that corresponds to the map and the number of agents of the given MAPF instance). Specifically, we run PP+ML with random restarts with a runtime limit of 3 seconds (i.e., repeatedly run PP using the stochastic ranking method until a solution is found or the runtime limit is reached) in each run and always use the same ML model for training and testing (i.e., train ML-T with datasets partially generated by ML-T and train ML-P with datasets partially generated by ML-P). This is effective in gathering training data for large numbers of agents on all small maps except for the warehouse map, for which we did not use PP+ML to generate training datasets (because it did not result in higher success rates). We varied the group size $m \in \{1, 5, 10\}$ for ML-T and picked $m = 5$ as it leads to the best results. We varied the regularization parameter $C \in \{0.1, 1, 10, 20, 100\}$ and picked $C = 20$ to train ML-T since there was no significant difference in the test results. We used the built-in cross-validation function in LIBLINEAR [52] to obtain the value of $C = 128$ to train ML-P.

We always train and test on the same grid map. For small grid maps, we train and test with the same number of agents. For large grid maps, we are only able to gather training datasets for MAPF instances with $k \leq 500$ because the runtime for a single PP run with a larger k is too high. We, therefore, train and test with the same number of agents when $k \leq 500$ and use the ML models trained on MAPF instances
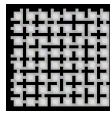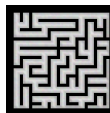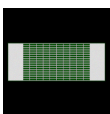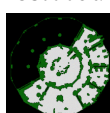
Grid Map	k	Success rate (%)					Solution rank				
		LH	SH	RND	ML-T	ML-P	LH	SH	RND	ML-T	ML-P
random	50	**96**	16	76	84	56	2.00	2.72	**1.24**	1.52	**1.24**
	100	**100**	20	60	32	24	1.20	1.68	**1.16**	1.32	1.68
	150	**68**	4	20	64	8	0.72	1.44	1.28	**0.68**	1.40
	200	24	0	0	**32**	24	0.44	0.80	0.80	**0.28**	0.44
	250	0	0	0	0	0	0.00	0.00	0.00	0.00	0.00
room	50	**88**	12	52	76	16	1.48	2.00	1.28	**0.72**	1.72
	75	**92**	0	8	0	44	**0.28**	1.44	1.36	1.44	0.72
	100	**60**	0	0	56	**60**	0.92	1.76	1.76	0.64	**0.56**
	125	**24**	0	0	16	20	**0.28**	0.60	0.60	0.44	0.32
	150	**4**	0	0	0	0	**0.00**	0.04	0.04	0.04	0.04
maze	50	**84**	0	12	76	68	1.32	2.40	2.12	**0.68**	0.96
	70	76	0	0	84	**88**	**0.84**	2.48	2.48	0.88	0.88
	90	64	0	0	52	**80**	0.96	1.96	1.96	0.84	**0.68**
	110	**44**	0	0	32	**44**	0.56	1.20	1.20	**0.52**	0.56
	130	**16**	0	0	8	**16**	**0.20**	0.40	0.40	0.32	**0.20**
warehouse	100	**92**	32	80	**92**	80	2.72	2.64	1.40	1.96	**0.64**
	200	**80**	28	56	36	60	1.80	1.36	1.56	1.44	**0.88**
	300	**52**	16	12	24	20	0.72	**0.68**	1.04	**0.68**	0.84
	400	**32**	4	8	12	24	**0.44**	0.64	0.64	0.60	**0.44**
	500	**12**	0	4	0	0	**0.04**	0.16	0.08	0.16	0.16
lak303d	300	**100**	28	96	96	76	2.64	2.64	1.96	**1.16**	1.24
	400	**100**	36	88	88	72	2.84	2.24	1.64	1.52	**1.08**
	500	**100**	44	88	84	80	2.76	1.76	1.48	2.24	**0.96**
	600	**88**	8	36	80	12	1.24	1.84	1.36	**0.72**	1.80
	700	16	0	0	**68**	0	0.64	0.84	0.84	**0.16**	0.84
ost003d	300	**100**	28	96	92	80	2.72	2.68	1.20	2.04	**1.04**
	400	88	32	**92**	84	**92**	2.84	2.44	2.04	1.04	**0.96**
	500	96	28	84	**96**	64	2.32	2.40	1.52	1.76	**1.20**
	600	**92**	16	60	84	40	1.76	2.16	**1.28**	1.32	1.44
	700	**72**	8	40	68	12	1.08	1.60	**0.88**	**0.88**	1.64

Table 2.12: Success rate and solution rank for deterministic ranking. The best results achieved among all algorithms are shown in bold. The results are obtained by training and testing on the same map with the same number of agents k, except for maps lak303d and ost003d with $k > 500$, where the results are obtained by training on the same map with $k = 500$.

with $k = 500$ to test on MAPF instances with $k > 500$. The runtime limit for testing is set to 1 minute for small grid maps and 10 minutes for large grid maps. We pre-compute the graph distances from each goal vertex to all vertices on the grid map and use them as the admissible heuristics for the space-time A* search for all variants of PP. Unlike Sections 2.6 and 2.8, we do not test on the grid map that is different from the one trained on since the ML models did not generalize well to unseen grid maps. For the same reason, we do not test on MAPF instances with numbers of agents that are different from the training instances.

We evaluate all variants of PP with four metrics on 25 test instances for each pair of number of agents and grid map. *Success rate* is the percentage of solved test instances within the runtime limit. *Runtime to first solution* is the runtime needed to find the first solution, averaged over all test instances, in which the runtime limit is used for unsolved instances. Here, we consider only the PP runtime and ignore the runtime overhead of generating the total priority orderings for PP because such runtime overhead for SH and LH is negligible as the start-goal graph distances are pre-computed and that for ML-T and ML-P are also small due to the small ML runtime overhead.** *Normalized sum of costs* is the ratio of the sum of costs and the sum of the start-goal graph distances of all agents. *Solution rank* evaluates the relative solution quality as follows: For each test instance, we rank the variants of PP in ascending order of the sums of costs of their solutions. The lower the sum of costs, the lower the numerical value of the rank. The lowest numerical value of the rank is 0. Algorithms that lead to the same sum of costs have the same rank, which is set to the numerical value of the lowest rank in the tie. For example, if the sums of costs of the 5 algorithms are 101, 101, 102, 103 and 103, then their ranks are 0, 0, 2, 3 and 3, respectively. Variants of PP that fail to solve the MAPF instances have the largest numerical value of the rank. Solution rank is the average numerical value of the rank over the test instances.

**The ML runtime overhead mainly comes from the runtime for collecting features, which, for example, is 0.03 seconds, 0.01 seconds, 0.02 seconds, 0.7 seconds, 4.37 seconds and 3.34 seconds per MAPF instance for the six grid maps with their respective largest numbers of agents tested in Table 2.13. Moreover, the features need to be collected only once for each MAPF instance even if we run PP multiple times via random restarts.
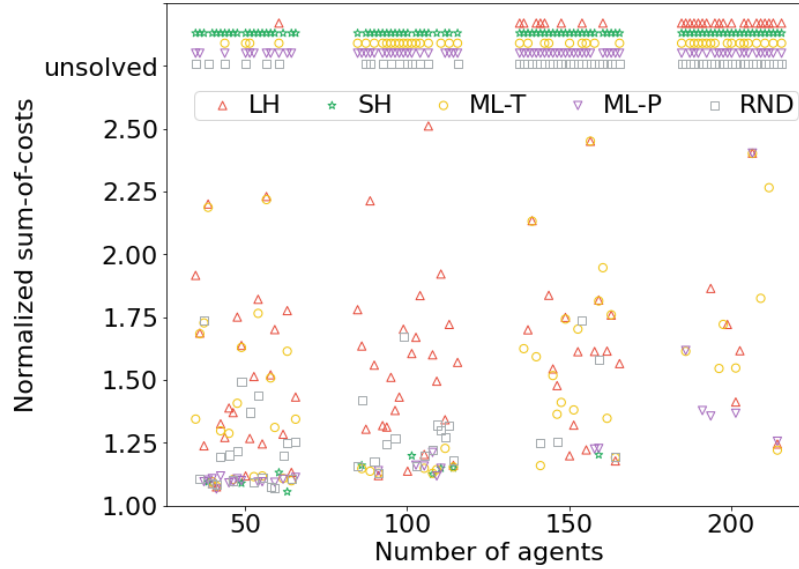
Figure 2.9: Normalized sum of costs for deterministic ranking on the random map. Unsolved MAPF instances are shown on top of the plot.

2.9.2.2 Results

Deterministic Ranking We first experiment with deterministic ranking for the baseline PP algorithms, LH, SH and RND and two variants of PP+ML, ML-T and ML-P, on test instances on each of the six grid maps and vary the number of agents. Here, each variant of PP generates one total priority ordering and runs exactly once for each MAPF instance. (RND uses the first total priority ordering that the randomized algorithm generates.) We report, in Table 2.12, the success rate and the solution rank for all grid maps and, in Figure 2.9, the normalized sum of costs for the random map.

In terms of success rate, both variants of PP+ML, ML-T and ML-P, achieve comparable results but do not completely dominate the non-ML-guided variants of PP. ML-T generally has a higher success rate than ML-P, but both are often more prone to failure than LH. In terms of solution quality, ML-T achieves results comparable to RND. Although ML-P often fails to find a solution, when it does find one, it often finds a solution with lower sum of costs than the other algorithms, as shown in Figure 2.9. In other words, ML-P suffers from low success rates but yields good solution qualities.
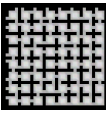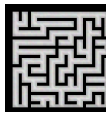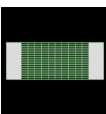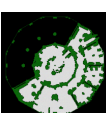
Grid Map	k	Success rate (%)					Runtime to the first solution (seconds)				
		LH	SH	RND	ML-T	ML-P	LH	SH	RND	ML-T	ML-P
random	150	100	100	100	100	100	**0.08**	0.65	0.42	0.24	1.37
	175	100	100	100	100	100	3.24	2.54	6.22	**1.06**	1.49
	200	88	80	88	**100**	**100**	15.60	20.56	18.25	**2.13**	2.86
	225	16	20	28	88	**92**	51.09	49.70	46.10	**8.82**	13.17
	250	0	0	0	44	**52**	60.00	60.00	60.00	40.88	**33.79**
room	50	100	100	100	100	100	0.24	0.17	0.15	**0.11**	0.65
	75	100	100	100	100	100	0.66	1.23	1.13	1.47	**0.52**
	100	84	80	76	**100**	**100**	12.56	22.70	23.07	3.35	**0.70**
	125	20	8	4	80	**88**	49.50	59.60	58.21	16.21	**10.18**
	150	0	0	0	24	**32**	60.00	60.00	60.00	51.53	**44.57**
maze	50	100	100	100	100	100	**0.61**	3.86	2.19	1.30	1.28
	70	**100**	68	68	96	**100**	3.17	25.48	28.58	3.24	**0.49**
	90	68	16	16	**100**	**100**	22.81	55.27	54.18	2.84	**0.72**
	110	44	0	0	92	**96**	33.67	60.02	60.01	15.22	**9.90**
	130	12	0	0	36	**52**	52.85	60.01	60.02	42.78	**33.90**
warehouse	350	96	96	96	**100**	92	**9.67**	13.62	13.18	13.45	13.43
	400	80	**84**	72	**84**	76	26.00	25.24	26.80	**24.39**	29.06
	450	**68**	48	52	60	48	**35.80**	39.67	39.45	37.10	40.91
	500	24	28	20	20	**32**	52.86	49.96	52.50	53.06	**49.11**
	550	12	8	8	**24**	12	58.29	56.73	59.54	**56.11**	56.63
lak303d	500	**100**	96	**100**	**100**	**100**	**26.74**	65.87	62.98	43.78	98.97
	600	**100**	96	92	96	88	**58.18**	117.80	135.97	100.26	174.64
	700	**100**	96	88	88	84	**99.51**	140.22	230.52	257.10	270.98
	800	**100**	68	76	68	56	**192.71**	397.20	395.42	381.47	451.95
	900	**68**	32	32	36	16	**423.82**	565.70	536.69	541.53	584.13
ost003d	500	**100**	**100**	96	96	92	**15.68**	49.15	60.60	53.10	87.81
	600	**100**	96	96	96	92	**43.07**	85.94	93.92	94.57	136.83
	700	**96**	92	92	92	88	**77.55**	163.39	205.10	468.27	232.78
	800	**100**	84	84	92	72	**126.35**	287.04	299.36	263.53	369.30
	900	**88**	64	64	72	40	**273.00**	462.16	456.27	420.71	525.51

Table 2.13: Success rate and runtime to the first solution for stochastic ranking with random restarts. The best results achieved among all algorithms are shown in bold. The results are obtained by training and testing on the same grid map with the same number of agents k, except for grid maps lak303d and ost003d with $k > 500$, where the results are obtained by training on the same map with $k = 500$.

Stochastic Ranking with Random Restarts   The random restart technique has been shown to improve the success rate of PP by trying multiple priority assignments [16]. Therefore, we use it to boost the success rate of both variants of PP+ML. We now illustrate stochastic ranking in conjunction with random restarts, which we apply to all five algorithms to ensure a fair comparison. To make random restarts possible, we add randomness to the deterministic algorithms LH and SH. LH relies on the start-goal graph distances of all agents to determine the total priority ordering. Therefore, for LH, we use the stochastic ranking method in Section 2.9.1.4 with $dist(s_i, g_i)$ replacing $\hat{y}_I(a_i)$ as agent a_i's score. For SH, since it is the reversed version of LH, we use the stochastic ranking method as for LH but generate a total priority ordering from low to high (instead of high to low). We varied parameter $\gamma \in \{0.1, 0.5, 1.0, 1.5\}$ in the softmax function and picked $\gamma = 0.5$ for SH, LH, ML-T and ML-P as it leads to the best results. RND is directly used with random restarts. We keep restarting each algorithm with a new random seed until the runtime limit is reached. We report, in Table 2.13, the success rate and the runtime to the first solution and, in Table 2.10, the solution rank, where the solution, which we refer to as the final solution, is the one with the least sum of costs found within the runtime limit.

ML-T and ML-P outperform all non-ML-guided variants of PP in terms of the success rate, runtime to the first solution and sum of costs of the final solution on the random, room and maze maps. ML-P has a slightly higher success rate and a better solution rank than ML-T. The advantage of PP+ML is most apparent on these grid maps when the number of agents is large and a solution is hard to find with the non-ML-guided variants of PP. On the warehouse map, ML-T and ML-P achieve results comparable to the baseline algorithms. On the large grid maps, ML-T achieves success rates and solution ranks comparable to the baseline algorithms, while ML-P has a marginally lower success rate but a better solution rank. These results, to some degree, are consistent with the difficulty of obtaining high-quality training datasets: As we described in the experimental setup at the beginning of this section, it is difficult to get good training datasets on the warehouse map and the large grid maps due to both the low success rates of existing PP
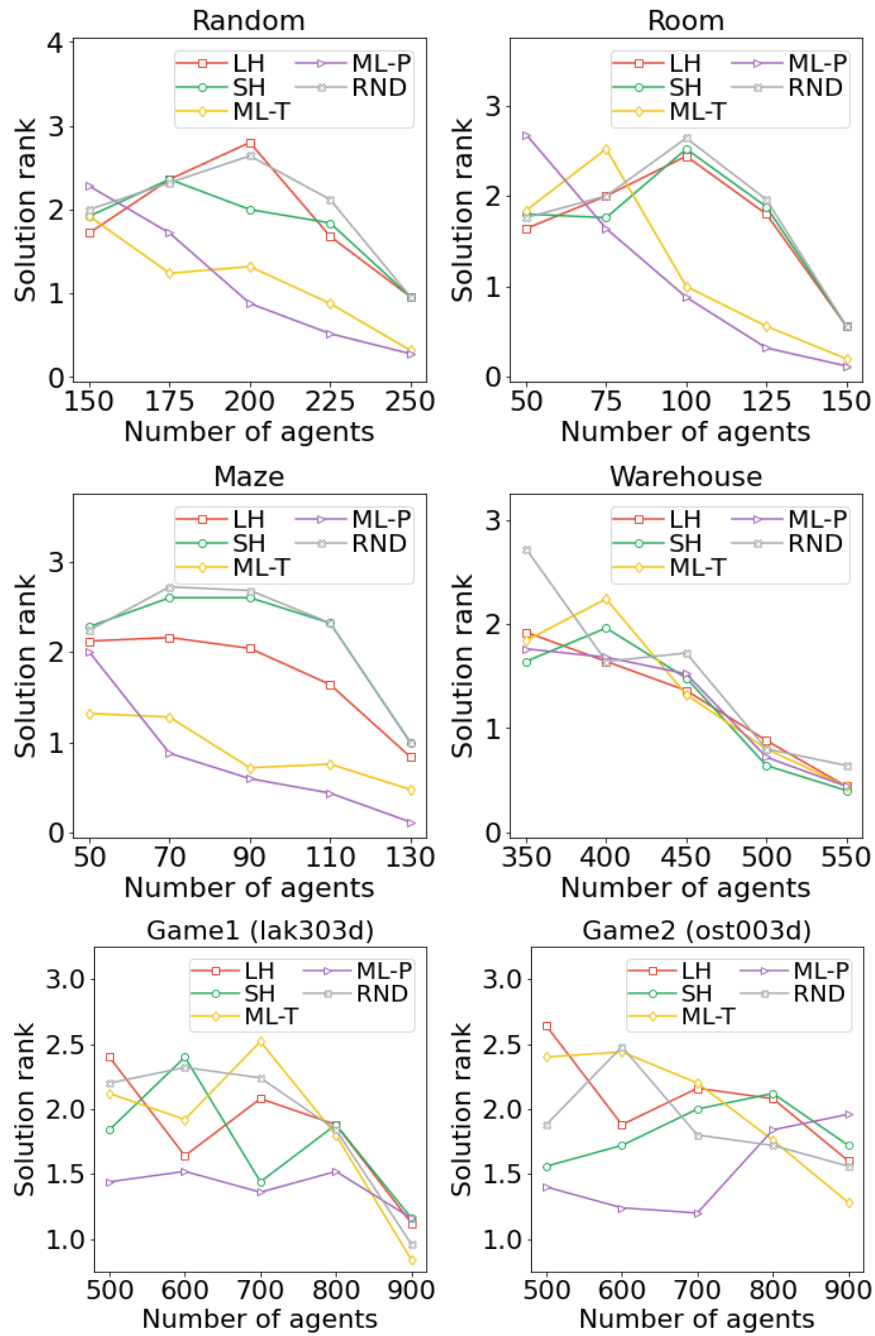
Figure 2.10: Solution rank for stochastic ranking with random restarts.

variants and their long runtimes. This is the reason why we train and test using different numbers of agents on the lak303d and ost003d maps. Our results on these grid maps demonstrate the limited ability of ML-P to generalize to a higher number of agents.

Feature Importance   We now analyze the feature importance of the learned ranking functions with good success rates and solution ranks, i.e., on the random, room and maze maps, each with the largest number of agents, because these ranking functions substantially outperform the baseline algorithms. We sort the feature weights w in decreasing order of their absolute values. Since the features are normalized, we use the absolute values of the feature weights to represent their importance.

The three ranking functions for ML-T, one for each grid map, have nine features in common among their top ten features with the largest absolute values: the graph and Manhattan distances between s_i and g_i and their absolute difference in Feature Group 1 (three features, definition in Section 2.9.1.2), the number of vertex conflicts counted by agent pair in Feature Group 11 (one feature) and Feature Groups 4, 6, 9, 10 and 12 (five features).

The three ranking functions for ML-P, one for each grid map, have five features in common among their top ten features with the largest absolute values: the graph distance between s_i and g_i and the absolute difference between the graph and Manhattan distances between s_i and g_i in Feature 1 (two features) and Feature Groups 4, 6 and 10 (three features).

Taking the intersection between the most important features for ML-T and ML-P, we determine the most important features to be Feature Groups 1, 4, 6 and 10, which correspond to the query-distance heuristic (Feature Group 1), the least-option heuristic (Feature Groups 4 and 6) and the start-and-goal-conflict heuristic (Feature Group 10). This indicates that our learned ranking functions cleverly combine the strengths of the existing heuristic methods.

2.10   Summary

In this chapter, we validated the hypothesis that one can leverage a general ML framework to improve human-designed decision-making strategies in different types of MAPF search algorithms. We first proposed a general ML framework based on imitation learning. To apply the framework, we find an expert to provide high-quality demonstrations of decisions that we are interested in improving and use the expert to collect data. Then, we learn an ML model to imitate the expert's decisions using imitation learning to speed up decision-making since the expert is slow. Finally, the learned ML model replaces the expert's decisions during the search.

We identified important decisions in CBS, ECBS, MAPF-LNS and PP, which are optimal, bounded-suboptimal and unbounded-suboptimal MAPF search algorithms, and then demonstrated the applicability of the framework to these algorithms. We introduced CBS+ML, ECBS+ML, MAPF-ML-LNS and PP+ML, where we learned an improved conflict-selection strategy for CBS, a node-selection strategy for ECBS, an agent-set selection strategy for MAPF-LNS and a priority-assignment strategy for PP that showed substantial improvement in empirical performance in terms of efficiency and/or effectiveness over their non-ML counterparts. Specifically, for CBS and ECBS, we improved their efficiency, and for MAPF-LNS and PP, we improved both their efficiency and effectiveness. With the imitation learning framework, we also showed how imitation learning with the same loss function, the same ML model and similar features can be reused in improving different MAPF search algorithms.

Chapter 3

Improving Decision-Making in MILP Search Algorithms

In this chapter, we present the second major contribution of this dissertation. In contrast to MAPF, imitation learning and reinforcement learning have been applied to improving MILP search algorithms. However, there are machine learning (ML) techniques, such as contrastive learning, that have shown success in various domains within computer science, such as computer vision [74] and natural language processing [66], but have not been applied to solving combinatorial optimization problems (COPs). To fill this gap, we formulate a general contrastive learning framework to improve decision-making strategies for MILP search algorithms. We identify important decisions to make in two different state-of-the-art MILP search algorithms, namely Large Neighborhood Search (LNS) and Predict-and-Search (PaS), and then apply the framework to improve them. Different from the imitation learning framework for MAPF introduced in Chapter 2, contrastive learning learns to make discriminative predictions based on the expert's demonstrations (that is, positive samples) and bad examples of demonstrations (that is, negative samples). One of the main challenges is to design algorithms to calculate both positive and negative samples, which is similar to finding an expert in the imitation learning framework. In this chapter, we will again see how the same ML algorithm, the same ML model, the same loss function and similar features can be reused in improving different MILP search algorithms. Empirically, the ML-guided versions of the MILP search algorithms substantially outperform their non-ML-guided and other ML-guided counterparts in terms of

both runtime and solution quality. Therefore, these results validate the hypothesis that one can leverage a general ML framework to improve human-designed decision-making strategies in different types of MILP search algorithms.

The remainder of this chapter is structured as follows. In Section 3.1, we state the motivation behind using ML for MILP solving and provide an overview of our contributions. In Section 3.2, we formally define MILPs. In Section 3.3, we introduce MILP search algorithms, including LNS and PaS. In Section 3.4, we summarize related work. In Section 3.5, we introduce the framework. In Sections 3.6 and 3.7, we introduce CL-LNS and ConPaS, respectively, and evaluate them empirically. Finally, in Section 3.8, we summarize the contributions of this chapter.

3.1   Introduction

Algorithm designs for COPs are important and challenging tasks. A wide variety of real-world problems are COPs, such as vehicle routing [189], path planning [158] and resource allocation [144] problems, and a majority of them are NP-hard to solve. In the past few decades, algorithms, including optimal algorithms, approximation algorithms and heuristic algorithms, have been studied extensively due to the importance of COPs. Those algorithms are mostly designed by humans through costly processes that often require a deep understanding of the problem domains and their underlying structures as well as considerable time and effort.

Recently, there has been an increased interest in automating algorithm designs for COPs with ML. Many ML methods learn to either construct or improve solutions or improve decision-making within an algorithmic framework, such as greedy search, local search or tree search, for a specific COP, such as MAPF, for which we demonstrate concrete examples in Chapter 2. Other examples include the traveling salesman problem (TSP) [200, 214], vehicle routing problem (VRP) [107] or independent set problem [130]. The ML methods for those COPs are often not easily applicable to the others.

In contrast, Mixed Integer Linear Programs (MILPs) can flexibly encode and solve a broad family of COPs, such as network design problems [96, 43, 84], mechanism design problems [41], facility location problems [76, 5]. MILPs can be solved by Branch and Bound (BnB) [113], an optimal tree search algorithm that can achieve state-of-the-art for MILPs. Over the past decades, BnB has been improved tremendously to become the core of many popular MILP solvers such as SCIP [21], CPLEX [37] and Gurobi [69]. However, due to its exhaustive search nature, it is hard for BnB to scale to large instances [102, 59].

On the other hand, meta-heuristic algorithms are MILP search algorithms that can find high-quality solutions much faster than BnB for large MILP instances. One of them is Large Neighborhood Search (LNS) [177, 198, 179, 86]. LNS starts from an initial solution (i.e., a feasible assignment of values to variables) and then improves the current best solution by iteratively selecting a subset of variables to reoptimize while leaving others fixed. Selecting which subset to reoptimize, i.e., the *destroy heuristic*, is a critical component in LNS. Hand-crafted destroy heuristics, such as the randomized heuristic [177, 179] and the Local Branching (LB) heuristic [56], are often either inefficient (slow to find good subsets) or ineffective (find subsets of bad quality). ML-based destroy heuristics have also been proposed and outperformed hand-crafted ones. State-of-the-art methods include IL-LNS [179] that uses imitation learning to imitate the LB heuristic and RL-LNS [198] that uses a similar framework to IL-LNS but trained with reinforcement learning.

Another line of research on meta-heuristic algorithms focuses on primal heuristics that generate high-quality solutions to MILPs. In particular, they focus on generating full or partial high-quality feasible assignments of values to variables. *Diving* is one of the most popular primal heuristics. In BnB, diving typically explores the BnB search tree to sequentially fix the values of the variables via depth-first search. Recently, there has been an increased interest in data-driven primal heuristic designs for MILPs since MILPs from the same application domain often share similar structures and characteristics. Among them, variants of diving [148, 70] have been proposed with a few main differences from diving: First, diving

100

can be performed at any search tree node in BnB and descend into the search tree to make assignments for variables sequentially, but the variants we discuss here make assignments for multiple variables all at once at the root node. Second, diving typically makes assignments for all variables, but these variants make assignments for only a subset of variables and then solve for the remaining variables with a MILP solver. One of these variants is called Neural Diving (ND) [148], where it learns to partially assign values to integer variables via imitation learning and delegate the reduced sub-MILP to a MILP solver, e.g., SCIP. The fraction of variables to assign values to is controlled by a hyperparameter called the coverage rate. A SelectiveNet [60] is trained for each coverage rate that jointly decides which variables to fix and the values to fix to during testing. The main two disadvantages of ND are that (1) enforcing variables to fixed values leads to low-quality or infeasible solutions if the predictions are not accurate enough and (2) it requires training multiple SelectiveNet to obtain the appropriate coverage rate, which is computationally expensive. To mitigate these issues, [70] propose another variant called Predict-and-Search (PaS) that deploys a search inspired by the trust region method. Instead of fixing variables, PaS searches for high-quality solutions within a pre-defined proximity of the predicted partial assignment, which allows better feasibility and finding higher-quality solutions than ND. For both ND and PaS, the crucial decisions to make are which variables to make assignment to and what values they should be assigned to. Their effectiveness (i.e., the quality of the solution found) and efficiency (i.e., the speed at which high-quality solutions are found) depend on the accuracy of the machine learning prediction and the number of variables (controlled by hyperparameters) whose values to fix.

We have mentioned important decision-making in two MILP search algorithms, namely, which subset of variables to select to reoptimize in LNS and deciding what values to assign to which subset of variables in PaS. In the past, ML methods that have been applied to improve them are mostly based on imitation learning [179, 177, 148, 70] or reinforcement learning [198, 177]. In this chapter, we propose a general

contrastive learning (CL) [30, 103] framework to learn such strategies and demonstrate that the performance of MILP search algorithms, i.e., the runtime and solution quality, can be improved with ML-guided strategies. CL is an ML method that enhances the performance of ML-guided strategies by contrasting good and bad decision samples to learn attributes that are common among good decisions and attributes that set apart good decisions from bad ones. In particular, we introduce CL-LNS and ConPaS to show that it is applicable to both LNS and PaS. To apply this CL framework to algorithms for MILP solving, we first identify an important decision to make in the search. Then, we collect training data. The crucial step in data collection for CL is to design both positive and negative samples representing good and bad decisions, respectively. By contrasting positive and negative samples, CL learns to make discriminative predictions of the decisions. Empirically, we show that variants of MILP search algorithms with contrastive-learned strategies substantially outperform their imitation-learned and/or reinforcement-learned counterparts in terms of both runtime and solution quality. The results also demonstrate how a general CL framework can be applied to advance state-of-the-art MILP search algorithms, which provide useful guidance to improve ML-guided MILP solving.

3.2 Mixed Integer Linear Programs

A *mixed integer linear program (MILP)* $M = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, q)$ is defined as

$$\min \boldsymbol{c}^\mathsf{T} \boldsymbol{x}$$

$$\text{s.t. } \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \tag{3.1}$$

$$\boldsymbol{x} \in \{0, 1\}^q \times \mathbb{R}^{n-q},$$

where $\boldsymbol{x} = (x_1, \ldots, x_n)^\mathsf{T}$ denotes the q binary variables and $n - q$ continuous variables to be optimized, $\boldsymbol{c} \in \mathbb{R}^n$ is the vector of objective coefficients, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{b} \in \mathbb{R}^m$ specify m linear constraints. A

Algorithm 7 LNS for MILPs

1: **Input:** A MILP M.
2: $\boldsymbol{x}^0 \leftarrow$ Find an initial solution to M
3: $t \leftarrow 0$
4: **while** time limit not exceeded **do**
5:     $\mathcal{X}^t \leftarrow$ Select a subset of binary variables to destroy
6:     $\boldsymbol{x}^{t+1} \leftarrow$ Solve the MILP M with additional constraints $\{x_i = x_i^t : i \leq q \wedge x_i \notin \mathcal{X}^t\}$
7:     $t \leftarrow t + 1$
8: **return** $\boldsymbol{x}^t$
---

solution $\boldsymbol{x}$ is *feasible* if its satisfies all the constraints. Finding an optimal solution to the MILP is NP-hard. In this chapter, for the purpose of demonstrating our methodologies, we focus on the mixed-binary formulation above. However, both our methods CL-LNS and ConPaS can also handle general integers using the same engineering techniques introduced in [148] and [179].

Linear Program (LP) Relaxation of a MILP   If we replace the integer constraints in Equations 3.1 with $\boldsymbol{x} \in [0,1]^q \times \mathbb{R}^{n-q}$, we obtain the linear program (LP) relaxation of the MILP. Finding an optimal solution to the LP relaxation takes polynomial time. The optimal solution to the LP relaxation is a lower bound of the MILP. If the optimal solution satisfies the integer constraints, it is also an optimal solution to the MILP.

3.3   Background

In this section, we provide detailed introductions to LNS for MILP solving, Neural Diving [148] and Predict-and-Search [70].

3.3.1   LNS for MILP solving

LNS is a heuristic algorithm that starts with an initial solution and then iteratively destroys and reoptimizes a part of the solution until a runtime limit is exceeded or some stopping condition is met. Let $M = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, q)$ be the input MILP, where $\boldsymbol{A}, \boldsymbol{b}$ and $\boldsymbol{c}$ are the coefficients and q is the number of binary variables defined in Equation (3.1), and $\boldsymbol{x}^0$ be the initial solution (typically found by running BnB for a short runtime).

In iteration $t \geq 0$ of LNS, given the *incumbent solution* $\boldsymbol{x}^t$, defined as the best solution found so far, a *destroy heuristic* selects a subset of k^t binary variables $\mathcal{X}^t = \{x_{i_1}, \ldots, x_{i_{k^t}}\}$. The reoptimization is done by solving a sub-MILP with $\mathcal{X}^t$ being the variables while fixing the values of $x_j \notin \mathcal{X}^t$ to the same values as in $\boldsymbol{x}^t$. The solution to the sub-MILP is the new incumbent solution $\boldsymbol{x}^{t+1}$ and then LNS proceeds to iteration $t + 1$. Compared to BnB, LNS is more effective in improving the objective value $\boldsymbol{c}^\mathsf{T} x$, especially on difficult and large-scale instances [177, 179, 198]. Compared to other local search methods, LNS explores a large neighborhood in each step and thus, is more effective in avoiding local minima. LNS for MILPs is summarized in Algorithm 7.

Adaptive Neighborhood Size   Adaptive methods are commonly used to set the neighborhood size k^t in previous work [179, 86]. The initial neighborhood size k^0 is set to a constant or a fraction of the number of binary variables. In this chapter, we consider the following adaptive method [86]: in iteration t, if LNS finds an improved solution, we let $k^{t+1} = k^t$, otherwise $k^{t+1} = \min\{\gamma \cdot k^t, \beta \cdot n\}$ where $\gamma > 1$ is a constant and we upper bound k^t to a constant fraction $\beta < 1$ of the number of binary variables to make sure the sub-MILP is not too large (thus, too difficult) to solve. Adaptively setting k^t helps LNS escape local minima by expanding the search neighborhood when it fails to improve the solution.

3.3.1.1   Local Branching Heuristic

The LB Heuristic [56] is originally proposed as a primal heuristic in BnB but also applicable in LNS for MILP solving [179, 131]. Given the incumbent solution $\boldsymbol{x}^t$ in iteration t of LNS, LB aims to find the subset of binary variables to destroy $\mathcal{X}^t$ such that it leads to the optimal $\boldsymbol{x}^{t+1}$ that differs from $\boldsymbol{x}^t$ on at most k^t variables, i.e., it computes the optimal solution $\boldsymbol{x}^{t+1}$ that sits within a given Hamming ball of radius k^t

centered around $\boldsymbol{x}^t$. To find $\boldsymbol{x}^{t+1}$, the LB heuristic solves the LB MILP that is exactly the same MILP from input but with one additional constraint that limits the distance between $\boldsymbol{x}^t$ and $\boldsymbol{x}^{t+1}$:

$$\sum_{i \leq q : x_i^t = 0} x_i^{t+1} + \sum_{i \leq q : x_i^t = 1} (1 - x_i^{t+1}) \leq k^t.$$

The LB MILP is of the same size of the input MILP (i.e., it has the same number of variables and one more constraint), therefore, it is often too slow to be useful in practice.

3.3.1.2 Local Branching Relaxation Heuristic

We propose the Local Branching Relaxation (LB-RELAX) heuristic in [86] that first solves the LP relaxation of the LB MILP and then selects variables $\mathcal{X}^t$ to destroy based on the LP relaxation solution. Specifically, given an MILP and the incumbent solution $\boldsymbol{x}^t$ in iteration t, we construct the LB MILP with neighborhood size k^t and solve its LP relaxation. Let $\bar{\boldsymbol{x}}^{t+1}$ be the LP relaxation solution to the LB MILP. Also, let $\Delta_i^t = |\bar{x}_i^{t+1} - x_i^t|$ and $\bar{\mathcal{X}}^t = \{x_i : \Delta_i^t > 0, i \leq q\}$. To construct $\mathcal{X}^t$ (the set of variables to destroy), LB-RELAX greedily selects k^t variables with the largest Δ_i^t from $\bar{\mathcal{X}}^t$ and breaks ties uniformly at random. If $\bar{\mathcal{X}}^t$ has less than k^t variables, we select all of them in $\bar{\mathcal{X}}^t$ and $k^t - |\bar{\mathcal{X}}^t|$ from the rest of the binary variables uniformly at random. Intuitively, LB-RELAX greedily selects the variables whose values are more likely to change in the incumbent solution $\boldsymbol{x}^t$ after solving the LB MILP. In [86], we propose two other variants of LB-RELAX with randomization. Empirically, LB-RELAX runs much faster than the LB heuristic. It also improves solutions faster than several state-of-the-art methods on a few problems but not for some others.

3.3.2 Neural Diving

ND [148] learns to generate a Bernoulli distribution for the solution values of binary variables. It learns the conditional distribution of the solution $\boldsymbol{x}$ given a MILP $M = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, q)$ defined as $p(\boldsymbol{x}|M) = \frac{\exp(-E(\boldsymbol{x}|M))}{\sum_{x' \in \mathcal{S}_\mathsf{p}^M} \exp(-E(\boldsymbol{x}'|M))}$, where $\mathcal{S}_\mathsf{p}^M$ is a set of optimal or near-optimal solutions to M and $E(\boldsymbol{x}|M)$ is an

energy function of a solution $\boldsymbol{x}$ defined as $\boldsymbol{c}^\mathsf{T}\boldsymbol{x}$ if $\boldsymbol{x}$ is feasible or ∞ otherwise. ND learns $\boldsymbol{p_\theta}(\boldsymbol{x}|M)$ parameterized by a graph convolutional network to approximate $p(\boldsymbol{x}|M)$ assuming conditional independence between variables $p(\boldsymbol{x}|M) \approx \prod_{i \leq q} p_\theta(x_i|M)$. Since the full prediction $\boldsymbol{p_\theta}(\boldsymbol{x}|M)$ might not give a feasible solution, ND predicts only a partial solution controlled by the coverage rates and employs SelectiveNet [60] to learn which variables' values to predict for each coverage rates. ND uses binary cross-entropy loss combined with the loss function for SelectiveNet to train the neural network. During testing, the input MILP M is then reduced to solving a smaller MILP after fixing the selected variables.

3.3.3 Predict-and-Search

Predict-and-Search (PaS) [70] uses the same framework as ND to learn to predict $p(\boldsymbol{x}|M)$. Instead of using SelectiveNet to learn to fix variables, PaS searches for near-optimal solutions within a neighborhood based on the prediction. Specifically, given the prediction $p_\theta(x_i|M)$ for each binary variable, PaS greedily selects k_0 binary variables $\mathcal{X}_0$ with the smallest $p_\theta(x_i|M)$ and k_1 binary variables $\mathcal{X}_1$ with the largest $p_\theta(x_i|M)$, such that $\mathcal{X}_0$ and $\mathcal{X}_1$ are disjoint ($k_0 + k_1 \leq q$). PaS fixes all variables in $\mathcal{X}_0$ to 0 and $\mathcal{X}_1$ to 1 in the sub-MILP, but also allows $\Delta \geq 0$ of the fixed variables to be flipped when solving it. Formally, let $B(\mathcal{X}_0, \mathcal{X}_1, \Delta) = \{\boldsymbol{x} : \sum_{x_i \in \mathcal{X}_0} x_i + \sum_{x_i \in \mathcal{X}_1} 1 - x_i \leq \Delta\}$ and D be the feasible region of the original MILP, PaS solves the following optimization problem:

$$\min \boldsymbol{c}^\mathsf{T}\boldsymbol{x} \quad \text{s.t. } \boldsymbol{x} \in D \cap B(\mathcal{X}_0, \mathcal{X}_1, \Delta). \tag{3.2}$$

Restricting the solution space to $B(\mathcal{X}_0, \mathcal{X}_1, \Delta)$ can be seen as a generalization of the fixing strategy employed in ND where $\Delta = 0$. Though in ND, $\mathcal{X}_0$ and $\mathcal{X}_1$ are constructed using sampling methods based on the neural network output.

3.4   Related Work

In this section, we summarize related work on LNS for MILPs and other COPs, LNS-based primal heuristics in BnB, learning to solve MILPs with BnB, solution predictions for COPs and contrastive learning for COPs.

3.4.1   LNS for MILPs and Other COPs

A huge effort has been made to improve BnB for MILPs in the past decades, but LNS for MILPs has not been studied extensively. Recently, [177] show that even a randomized destroy heuristic in LNS can outperform state-of-the-art BnB. They also show that an ML-guided decomposition-based LNS can achieve even better performance, where they apply reinforcement learning and imitation learning to learn destroy heuristics that decompose the set of variables into equally-sized subsets using a classification loss. [179] learn to select variables by imitating LB. RL-LNS [198] uses a similar framework but trained with reinforcement learning and outperforms [177]. Both [198] and [179] use the bipartite graph representations of MILPs to learn the destroy heuristics represented by GCNs. Another line of related work focuses on improving the LB heuristic. [131] use ML to tune the runtime limit and neighborhood sizes for LB. [86] propose LB-RELAX to select variables by solving the LP relaxation of LB.

Besides MILPs, LNS has been applied to solve many COPs, such as VRP [166, 8], TSP [176], scheduling [108, 215] and MAPF [119, 117, 91]. ML methods have also been applied to improve LNS for those applications [32, 133, 82, 127, 90].

3.4.2   LNS-Based Primal Heuristics in BnB

LNS-based primal heuristics are a family of primal heuristics in BnB and have been studied extensively. With the same purpose of improving primal bounds, the main differences between the LNS-based primal heuristics in BnB and LNS for MILPs are: (1) LNS-based primal heuristics are executed periodically at different search tree nodes during the search and the execution schedule is itself dynamic because they

are often more expensive to run than the other primal heuristics in BnB; (2) the destroy heuristics in LNS-based primal heuristics are often designed to use information specific to BnB, such as the dual bound and the LP relaxation at a search tree node, and they are not directly applicable in LNS for MILPs in our setting.

Next, we briefly summarize the destroy heuristics in LNS-based primal heuristics:

- **Crossover Heuristics [169]** It destroys variables that have different values in a set of selected known solutions (typically two).

- **Mutation heuristics [169]** It destroys a random subset of variables.

- **Relaxation Induced Neighborhood Search [39]** It destroys variables whose values disagree in the solution of the LP relaxation at the search tree node and the incumbent solution.

- **Relaxation Enforced Neighborhood Search [20]** It restricts the neighborhood to be the feasible roundings of the LP relaxation at the current search tree node.

- **Local Branching [56]** It restricts the neighborhood to a ball around the current incumbent solution.

- **Distance Induced Neighborhood Search [61]** It takes the intersection of the neighborhoods of the Crossover, Local Branching and Relaxation Induced Neighborhood Search heuristics.

- **Graph-Induced Neighborhood Search [142]** It destroys the breadth-first-search neighborhood of a variable in the bipartite graph representation of the MILP.

Recently, an adaptive LNS primal heuristic [75] has been proposed to combine the power of these heuristics, where it essentially solves a multi-armed bandit problem to choose which heuristic to apply.

3.4.3 Learning to Solve MILPs with BnB

Several studies have applied ML to improve BnB. The majority of works focus on learning to either select variables to branch on [102, 59, 68, 211] or select nodes to expand [73, 110]. There are also works on learning to schedule and run primal heuristics [100, 34] and to select cutting planes [184, 154, 92].

3.4.4 Solution Predictions for COPs

There are other works on learning to predict solutions to MILPs in addition to ND and PaS. [44] learn to predict backbone variables [48] whose values stay unchanged across different optimal and near-optimal solutions and then search for optimal solutions based on the predicted backbone variables. However, this method is not applicable to many COPs since backbone variables do not necessarily exist for them. Recently, [203] propose threshold-aware learning to optimize the coverage rate in ND and is one of the state-of-the-art methods. However, this method also fixes variables when solving the sub-MILP. [101] and [130] learn to guide decision-making, such as warm-starting and node selection, in COP solvers, such as MIP solvers and local search, via solution predictions.

3.4.5 Contrastive Learning for COPs

While contrastive learning of visual representations [77, 74, 30] and graph representations [205, 188] have been studied extensively, it has not been explored much for COPs. [147] derive a contrastive loss for decision-focused learning to solve COPs with uncertain inputs that can be learned from historical data, where they view non-optimal solutions as negative samples. [46] use contrastive pre-training to learn good representations for the boolean satisfiability problem.

3.5 A Contrastive Learning Framework for Learning Decision-Making Strategies

In this section, we introduce a general ML framework based on contrastive learning to learn decision-making strategies for MILP solving. As will be shown in Sections 3.6 and 3.7, this is a framework that has been successfully applied to different tasks for MILP solving. We employ CL rather than other learning methods, such as imitation learning and reinforcement learning, because it has been theoretically demonstrated to be effective [187]. CL has empirically outperformed them in combinatorial optimization problems [46, 147] and other problem domains [51]. The framework consists of the following steps:

1. **Identify a Decision to Improve** Given a MILP search algorithm, identify a decision that is crucial to its performance. The goal is to learn a strategy to improve making this decision.

2. **Data Collection** One of the crucial steps in CL is to design both positive and negative samples. Similar to imitation learning, CL learns from positive samples, which are high-quality demonstrations of the decisions and can be acquired from an expert. Unlike imitation learning, CL requires negative samples, which are low-quality or infeasible demonstrations of the decisions. It is encouraged to find negative samples that are deceptively similar to positive ones since it has been analyzed to be beneficial for CL [187]. For features, one of the popular techniques to featurize MILPs is using its bipartite graph representation [59], which is often used with a graph neural network. In this chapter, we use such an engineering technique, but this framework is compatible with the others, such as those introduced in [102] and [177].

3. **Model Learning with a Contrastive Loss** The goal is to learn a model to predict decisions that are as similar to the positive samples as possible and, at the same time, dissimilar to the negative samples. A contrastive loss is a function whose value is low when this holds true. In this chapter,
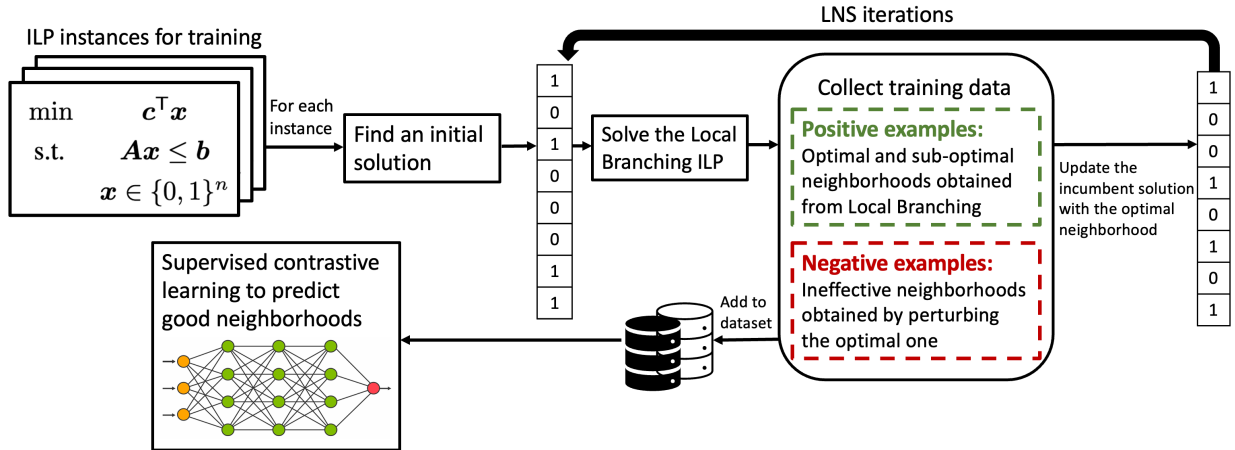
Figure 3.1: An overview of training and data collection for CL-LNS. For each MILP instance for training, we run several LNS iterations with LB. In each iteration, we collect both positive and negative neighborhood samples and add them to the training dataset, which is used in downstream supervised contrastive learning for neighborhood selections.

we utilize a form of supervised contrastive loss, called InfoNCE [151, 74], but this framework is compatible with other contrastive losses, such as the margin loss [195] and the triplet loss [160].

4. **ML-Guided Search** Once we have a trained ML model, we plug it into the MILP search algorithm as a decision-making strategy.

3.6 Contrastive Large Neighborhood Search

In this section, we introduce CL-LNS to show how the framework can be applied to learn efficient and effective destroy heuristics. Similar to IL-LNS [179], we learn to imitate the LB heuristic, a destroy heuristic that selects the optimal subset of variables within the Hamming ball of the incumbent solutions. LB requires solving another MILP with the same size as the original problem and thus is computationally expensive. We not only use the optimal subsets provided by LB as the expert demonstration (as in IL-LNS) but also leverage intermediate solutions and perturbations. When solving the MILP for LB, intermediate solutions are found and those that are close to optimal in terms of effectiveness become positive samples. We also collect negative samples by randomly perturbing the optimal subsets. With both positive and

negative samples, instead of a classification loss as in IL-LNS, we use a contrastive loss that encourages the model to predict the subset similar to the positive samples but dissimilar to the negative ones with similarity measured by dot products [151, 74]. Finally, we also use a richer set of features and graph attention networks (GAT) instead of GCN to further boost performance. Empirically, we show that CL-LNS outperforms state-of-the-art ML-guided and non-ML-guided versions of LNS at different runtime cutoffs ranging from a few minutes to an hour in terms of multiple metrics, including the primal gap, the primal integral, the best performing rate and the survival rate, demonstrating the effectiveness and efficiency of CL-LNS. In addition, CL-LNS shows great generalization performance on test instances 100% larger than training instances.

3.6.1 Machine Learning Methodology

Our goal is to learn a policy, a destroy heuristic represented by an ML model, that selects a subset of variables to destroy and reoptimize in each LNS iteration. Specifically, let $\boldsymbol{s}^t = (M, \boldsymbol{x}^t)$ be the current state in iteration t of LNS where $M = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, q)$ is the MILP and $\boldsymbol{x}^t$ is the incumbent solution, the policy predicts an action $\boldsymbol{a}^t = (a_1^t, \ldots, a_q^t) \in \{0, 1\}^q$, a binary representation of the selected binary variables $\mathcal{X}^t$ indicating whether x_i is selected ($a_i^t = 1$) or not ($a_i^t = 0$). We use contrastive learning to learn to predict high quality $\boldsymbol{a}^t$ such that, after solving the sub-MILP derived from $\boldsymbol{a}^t$ (or $\mathcal{X}^t$), the resulting incumbent solution $\boldsymbol{x}^{t+1}$ is improved as much as possible. Next, we describe our novel data collection process, the policy network and the contrastive loss used in training. An overview of our training and data collection pipeline is shown in Figure 3.1. Finally, we introduce how the learned policy is used in CL-LNS.

3.6.1.1 Data Collection

Following previous work [179], we use LB as the expert policy to collect good demonstrations to learn to imitate. Formally, for a given state $\boldsymbol{s}^t = (M, \boldsymbol{x}^t)$, we use LB to find the optimal action $\boldsymbol{a}^t$ that leads to the

minimum $c^\mathsf{T} x^{t+1}$ after solving the sub-MILP. Different from previous work [179, 177], we use contrastive learning to learn to make discriminative predictions of a^t by contrasting positive and negative samples (i.e., good and bad examples of actions a^t). In the following, we describe how we collect the positive sample set $\mathcal{S}_\mathsf{p}^t$ and the negative sample set $\mathcal{S}_\mathsf{n}^t$.

Collecting Positive Samples $\mathcal{S}_\mathsf{p}^t$ During data collection, given $s^t = (M, x^t)$, we solve the LB MILP with the incumbent solution x^t and neighborhood size k^t to find the optimal x^{t+1}. LNS proceeds to iteration $t + 1$ with x^{t+1} until no improving solution x^{t+1} could be found by the LB MILP within a runtime limit. In experiments, the LB MILP is solved with SCIP 8.0.1 [21] with an hour runtime limit and k^t is fine-tuned for each type of instances. After each solve of the LB MILP, in addition to the best solution found, SCIP records all intermediate solutions found during the solve. We look for intermediate solutions x' whose resulting improvements on the objective value is at least $0 < \alpha_\mathsf{p} \leq 1$ times the best improvement (i.e., $c^\mathsf{T}(x^t - x') \geq \alpha_\mathsf{p} \cdot c^\mathsf{T}(x^t - x^{t+1})$) and consider their corresponding actions as positive samples. We limit the number of the positive samples $|\mathcal{S}_\mathsf{p}^t|$ to u_p. If more than u_p positive samples are available, we record the top u_p ones to avoid large computational overhead with too many samples when computing the contrastive loss (see subsection 3.6.1.3). α_p and u_p are set to 0.5 and 10, respectively, in experiments.

Collecting Negative Samples $\mathcal{S}_\mathsf{n}^t$ Negative samples are critical parts of contrastive learning to help distinguish between good and bad demonstrations. We collect a set of c_n^t negative samples $\mathcal{S}_\mathsf{n}^t$, where $c_\mathsf{n}^t = \kappa |\mathcal{S}_\mathsf{p}^t|$ and κ is a hyperparameter to control the ratio between the numbers of positive and negative samples. Suppose $\mathcal{X}^t$ is the optimal set of variables selected by LB. We then perturb $\mathcal{X}^t$ to get $\hat{\mathcal{X}}^t$ by replacing 5% of the variables in $\mathcal{X}^t$ with the same number of those binary variables not in $\mathcal{X}^t$ uniformly at random. We then solve the corresponding sub-MILP derived from $\hat{\mathcal{X}}^t$ to get a new incumbent solution $\hat{x}^{t+1}$. If the resulting improvement of $\hat{x}^{t+1}$ is less than $0 \leq \alpha_\mathsf{n} < 1$ times the best improvement (i.e., $c^\mathsf{T}(x^t - \hat{x}^{t+1}) \leq \alpha_\mathsf{n} \cdot c^\mathsf{T}(x^t - x^{t+1})$), we consider its corresponding action as a negative sample. We

repeat this c_n^t times to collect negative samples. If less than c_n^t negative samples is collected, we increase the perturbation rate from 5% to 10% and generate another c_n^t samples. We keep increasing the perturbation rate at an increment of 5% until c_n^t negative samples are found or it reaches 100%. In experiments, we set $\kappa = 9$ and $\alpha_\mathsf{n} = 0.05$ and it takes less than 5 minutes to collect negative samples for each state.

3.6.1.2 Neural Network Architecture

Following previous work on learning for MILPs [59, 179, 198], we use a bipartite graph representation of MILP to encode a state $\boldsymbol{s}^t$. The bipartite graph consists of $n + m$ nodes representing the n variables and m constraints on two sides, respectively, with an edge connecting a variable and a constraint if the variable has a non-zero coefficient in the constraint. Following [179], we use features proposed in [59] for node features and edge features in the bipartite graph and also include a fixed-size window of most recent incumbent values as variable node features with the window size set to 3 in experiments. In addition to features used in [179], we include features proposed in [102] computed at the root node of BnB to make it a richer set of variable node features.

We learn a policy $\pi_{\boldsymbol{\theta}}(\cdot)$ represented by a GAT [23] parameterized by learnable weights $\boldsymbol{\theta}$. The policy takes as input the state $\boldsymbol{s}^t$ and outputs a score vector $\pi_{\boldsymbol{\theta}}(\boldsymbol{s}^t) \in [0, 1]^q$, one score per variable. To increase the modeling capacity and to manipulate node interactions proposed by our architecture, we use embedding layers to map each node feature and edge feature to space $\mathbb{R}^d$. Let $\mathbf{v}_j, \mathbf{c}_i, \mathbf{e}_{i,j} \in \mathbb{R}^d$ be the embeddings of the j-th variable, i-th constraint and the edge connecting them output by the embedding layers. Since our graph is bipartite, following previous work [59], we perform two rounds of message passing through the GAT. In the first round, each constraint node $\mathbf{c}_i$ attends to its neighbors $\mathcal{N}_i$ using an attention structure with H attention heads to get updated constraint embeddings $\mathbf{c}_i'$ (computed as a function of $\mathbf{v}_j, \mathbf{c}_i, \mathbf{e}_{i,j}$). In the second round, similarly, each variable node attends to its neighbors to get updated variable embeddings $\mathbf{v}'$ (computed as a function of $\mathbf{v}_j, \mathbf{c}_i', \mathbf{e}_{i,j}$) with another set of attention weights. After the two rounds of

message passing, the final representations of variables $\mathbf{v}'$ are passed through a multi-layer perceptron (MLP) to obtain a scalar value for each variable and, finally, we apply the sigmoid function to get a score between 0 and 1. Full details of the network architecture are provided in Appendix. In experiments, d and H are set to 64 and 8, respectively.

3.6.1.3 Model Learning with a Contrastive Loss

Given a set of MILP instances for training, we follow the expert's trajectory to collect training data. Let $\mathcal{D}^{\text{CL-LNS}} = \{(s, \mathcal{S}_{\mathsf{p}}, \mathcal{S}_{\mathsf{n}})\}$ be the set of states with their corresponding sets of positive and negative samples in the training data. A contrastive loss is a function whose value is low when the predicted action $\pi_{\boldsymbol{\theta}}(s)$ is similar to the positive samples $\mathcal{S}_{\mathsf{p}}$ and dissimilar to the negative samples $\mathcal{S}_{\mathsf{n}}$. With similarity measured by dot products, a form of supervised contrastive loss, called InfoNCE [151, 74], is used for CL-LNS:

$$\mathcal{L}^{\text{CL-LNS}}(\boldsymbol{\theta}) = \sum_{(s, \mathcal{S}_{\mathsf{p}}, \mathcal{S}_{\mathsf{n}}) \in \mathcal{D}^{\text{CL-LNS}}} \frac{-1}{|\mathcal{S}_{\mathsf{p}}|} \sum_{\boldsymbol{a} \in \mathcal{S}_{\mathsf{p}}} \log \frac{\exp(\boldsymbol{a}^{\mathsf{T}} \pi_{\boldsymbol{\theta}}(s)/\tau)}{\sum_{\boldsymbol{a}' \in \mathcal{S}_{\mathsf{n}} \cup \{\boldsymbol{a}\}} \exp(\boldsymbol{a}'^{\mathsf{T}} \pi_{\boldsymbol{\theta}}(s)/\tau)}$$

where τ is a temperature hyperparameter set to 0.07 [74] in experiments.

3.6.1.4 ML-Guided Search

During testing, we apply the learned policy $\pi_{\boldsymbol{\theta}}$ in LNS. In iteration t, let $(v_1, \cdots, v_q) := \pi_{\boldsymbol{\theta}}(s^t)$ be the variable scores output by the policy. To select k^t variables, CL-LNS greedily selects those with the highest scores. Previous works [179, 198] commonly use sampling methods to select the variables, but those sampling methods are empirically worse than our greedy method in CL-LNS. However, when the adaptive neighborhood size k^t reaches its upper bound $\beta \cdot q$, CL-LNS may repeat the same prediction due to the deterministic selection process. When this happens, we switch to the sampling method introduced in [179]. The sampling method selects variables sequentially: at each step, a variable x_i that has not been

	Small Instances				Large Instances			
Name	MVC-S	MIS-S	CA-S	SC-S	MVC-L	MIS-L	CA-L	SC-L
#Variables	1,000	6,000	4,000	4,000	2,000	12,000	8,000	8,000
#Constraints	65,100	23,977	2,675	5,000	135,100	48,027	5,353	5,000

Table 3.1: Names and the average numbers of variables and constraints of the test instances.

selected yet is selected with probability proportional to v_i^η, where η is a temperature parameter set to 0.5 in experiments.

3.6.2 Empirical Evaluation

In this subsection, we demonstrate the efficiency and effectiveness of CL-LNS through experiments. In the following, we introduce our evaluation setup and then present the results.

3.6.2.1 Setup

Instance Generation We evaluate on four NP-hard MILP problems that are widely used in existing studies [198, 177, 173], which consist of two graph optimization problems, namely the minimum vertex cover (MVC) and maximum independent set (MIS) problems, and two non-graph optimization problems, namely the combinatorial auction (CA) and set covering (SC) problems. We first generate 100 small test instances for each MILP problem, namely MVC-S, MIS-S, CA-S and SC-S. MVC-S instances are generated according to the Barabasi-Albert random graph model [3], with 1,000 nodes and an average degree of 70 following [177]. MIS-S instances are generated according to the Erdos-Renyi random graph model [50], with 6,000 nodes and an average degree of 5 following [177]. CA-S instances are generated with 2,000 items and 4,000 bids according to the arbitrary relations in [115]. SC-S instances are generated with 4,000 variables and 5,000 constraints following [198]. We then generate another 100 large test instances for each MILP problem by doubling the number of variables, namely MVC-L, MIS-L, CA-L and SC-L. For each set of test instances, Table 3.4 shows its average numbers of variables and constraints. More details of instance generation are included in Appendix.

For data collection and training, we generate another set of 1,024 small instances for each MILP problem. We split these instances into 892 training instances and 128 validation instances.

Baselines We compare CL-LNS with five baselines: (1) BnB: using SCIP (v8.0.1), the state-of-the-art open-source MILP solver, with the aggressive mode fine-tuned to focus on improving the objective value; (2) RANDOM: LNS which selects the neighborhood by uniformly sampling k^t variables without replacement; (3) LB-RELAX [86]: LNS which selects the neighborhood with the LB-RELAX heuristics; (4) IL-LNS [179]; (5) RL-LNS [198]. We compare with two more baselines in Appendix. For each ML method, a separate model is trained for each MILP problem on the small training instances and tested on both small and large test instances. We implement IL-LNS and fine-tune its hyperparameters for each MILP problem since the authors do not fully open-source the code. For RL-LNS, we use the code and hyperparameters provided by the authors and train the models with five random seeds to select one with the best performance on the validation instances. We do not compare to the method by [177] since it performs worse than RL-LNS on multiple MILP problems [198]. For both IL-LNS and RL-LNS, we also test their generalization performance on the large instances.

Metrics We use the following metrics to evaluate all methods:

1. The *primal bound* is the objective value of the MILP;

2. The *primal gap* [19] is the normalized difference between the primal bound v and a precomputed best known objective value v^*, defined as $\frac{|v-v^*|}{\max(v,v^*,\varepsilon)}$ if v exists and $v \cdot v^* \geq 0$, or 1 otherwise. We use $\varepsilon = 10^{-8}$ to avoid division by zero; v^* is the best primal bound found within 60 minutes by any method in the portfolio for comparison.

3. The *primal integral* [1] at time z is the integral on $[0, z]$ of the primal gap as a function of runtime. It captures the quality of and the speed at which solutions are found. This is similar to the area under the curve that we use for MAPF-LNS in Section 2.8.2;

4. The *survival rate* to meet a certain primal gap threshold is the fraction of instances with primal gaps below the threshold [179];

5. The *best performing rate* of a method is the fraction of instances on which it achieves the best primal gap (including ties) compared to all methods at a given runtime cutoff.

Hyperparameters   We conduct experiments on 2.5GHz Intel Xeon Platinum 8259CL CPUs with 32 GB memory. Training is done on a NVIDIA A100 GPU with 40 GB memory. All experiments use the hyper-parameters described below unless stated otherwise. We use SCIP (v8.0.1) [21] to solve the sub-MILP in every iteration of LNS. To run LNS, we find an initial solution by running SCIP for 10 seconds. We set the time limit to 60 minutes to solve each instance and 2 minutes to solve the sub-MILP in every LNS iteration. All methods require a neighborhood size k^t in LNS, except for BnB and RL-LNS (k^t in RL-LNS is defined implicitly by how the policy is used). For LB-RELAX, IL-LNS and CL-LNS, the initial neighborhood size k^0 is set to $100, 3000, 1000$ and 150 for MVC, MIS, CA and SC, respectively, except k^0 is set to 150 for SC for IL-LNS; for RANDOM, it is set to $200, 3000, 1500$ and 200 for MVC, MIS, CA and SC, respectively. All methods use adaptive neighborhood sizes with $\gamma = 1.02$ and $\beta = 0.5$, except for BnB and RL-LNS. For IL-LNS, when applying its learned policies, we use the sampling methods on MVC and CA instances and the greedy method on SC and MIS instances. For CL-LNS, the greedy method is used on all instances. Additional details on hyperparameter tunings are provided in Appendix.

For data collection, we use different neighborhood sizes $k^0 = 50, 500, 200$ and 50 for MVC, MIS, CA and SC, respectively, which we justify in subsection 3.6.2.2. We set $\gamma = 1$ and run LNS with LB until no new incumbent solution is found (i.e., we do not adaptively update neighborhood sizes during data

collection). The runtime limit for solving LB in every iteration is set to 1 hour. For training, we use the Adam optimizer [104] with learning rate 10^{-3}. We use a batch size of 32 and train for 30 epochs (the training typically converges in less than 20 epochs and 24 hours).

Since BnB and LNS are both anytime algorithms, we show these metrics as a function of runtime or the number of iterations in LNS (when applicable) to demonstrate their anytime performance.

3.6.2.2 Results

Figure 3.8 shows the primal gap as a function of runtime. Table 3.2 presents the average primal gap and primal integral at 60-minute runtime cutoff on small and large instances, respectively (see results at 30-minute runtime cutoff in Appendix). Note that we were not able to reproduce the results on CA-S and CA-L reported in [198] for RL-LNS despite using their code and repeating training with five random seeds. CL-LNS shows better anytime performance than all baselines on all MILP problems. On the small instances, it achieves 32%-42% lower average primal gaps and 26%-59% lower average primal integrals than the second-best method at the 60-minute runtime cutoff. It also demonstrates strong generalization performance on large instances unseen during training, reducing the second-best average primal gap and average primal integral by up to 94.4% and 57.1%, respectively. Figure 3.9 shows the survival rate to meet the 1.00% primal gap threshold. CL-LNS achieves the best survival rate at the 60-minute runtime cutoff on all instances, except that, on SC-L, its final survival rate is slightly worse than RL-LNS, but it achieves the rate with a much shorter runtime. On MVC-L, MIS-S and MIS-L instances, several baselines achieve the same survival rate as CL-LNS, but it always achieves the rates with the shortest runtime. Figure 3.4 shows the best performing rate. CL-LNS consistently performs best on 50% to 100% of the small instances and has the highest best performing rate in most cases on the large instances. In Appendix, we present strong results in comparison with two more baselines and on one more performance metric.
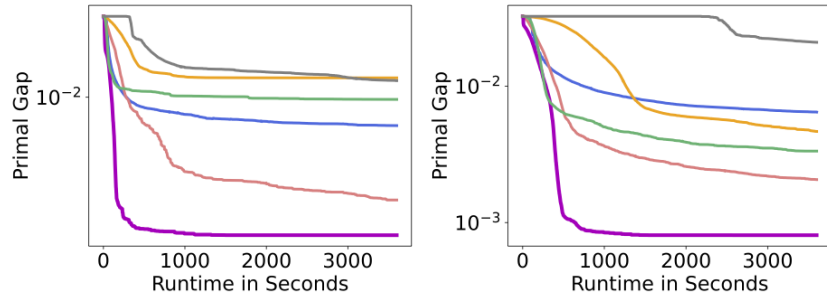
(a) MVC-S (left) and MVC-L (right).
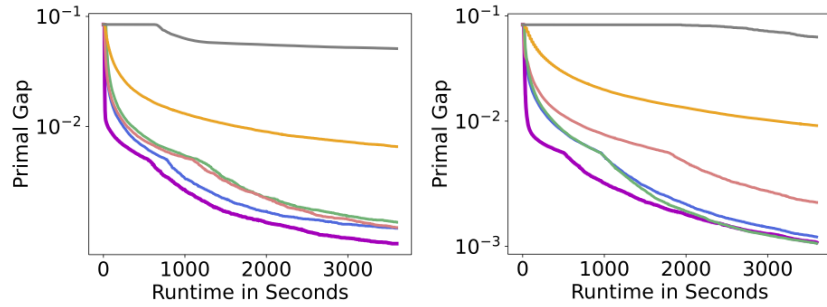
(b) MIS-S (left) and MIS-L (right).
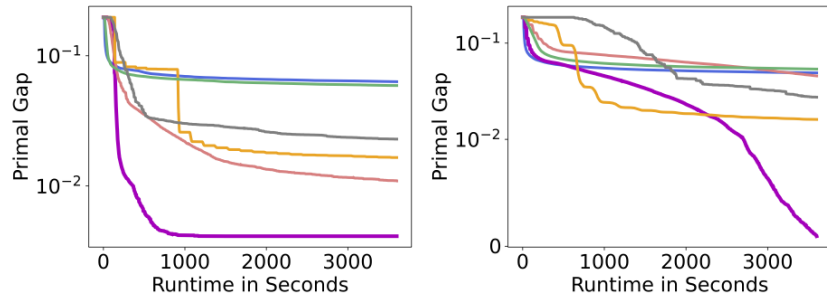
(c) CA-S (left) and CA-L (right).
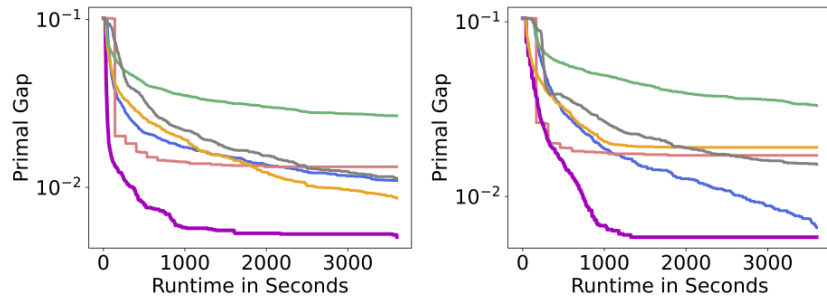
(d) SC-S (left) and SC-L (right).

Figure 3.2: The primal gap (the lower, the better) as a function of runtime averaged over 100 test instances. For ML methods, the policies are trained only on small training instances but are tested on both small and large test instances.

	PG (%) ↓	PI ↓	PG (%) ↓	PI ↓
	MVC-S		MIS-S	
BnB	1.32±0.43	66.1±13.1	5.10±0.69	222.8±25.9
RANDOM	0.96±1.26	38.0±44.8	0.24±0.14	22.1±5.0
LB-RELAX	1.38±1.51	57.0±51.2	0.65±0.20	46.9±6.5
IL-LNS	0.29±0.23	19.2±10.2	0.22±0.17	19.4±5.8
RL-LNS	0.61±0.34	29.6±11.5	0.22±0.14	17.2±5.2
CL-LNS	**0.17±0.09**	**8.7±6.7**	**0.15±0.15**	**12.8±5.4**
	CA-S		SC-S	
BnB	2.28±0.59	137.4±25.9	1.13±0.95	86.7±37.9
RANDOM	5.90±1.02	235.6±34.9	2.67±1.29	124.3±45.4
LB-RELAX	1.65±0.57	140.5±18.3	0.86±0.83	63.2±31.6
IL-LNS	1.09±0.51	90.0±20.8	1.33±0.97	63.2±34.3
RL-LNS	6.32±1.03	249.2±35.9	1.10±0.77	77.8±28.9
CL-LNS	**0.65±0.32**	**50.7±22.7**	**0.50±0.58**	**26.2±12.8**
	MVC-L		MIS-L	
BnB	2.41±0.40	130.2±11.1	6.29±1.62	285.1±18.2
RANDOM	0.38±0.24	22.7±8.0	**0.11±0.08**	19.0±3.1
LB-RELAX	0.46±0.23	48.4±7.5	0.91±0.16	68.6±5.5
IL-LNS	0.27±0.23	21.2±8.1	0.29±0.15	27.1±5.5
RL-LNS	0.59±0.30	37.3±9.6	0.14±0.12	18.9±4.1
CL-LNS	**0.05±0.04**	**9.1±3.4**	0.12±0.11	**12.9±4.4**
	CA-L		SC-L	
BnB	2.74±1.87	320.9±83.1	1.54±1.33	115.0±42.5
RANDOM	5.37±0.75	229.2±24.4	3.31±1.79	166.4±61.3
LB-RELAX	1.61±1.50	153.0±50.3	1.91±1.42	88.3±48.9
IL-LNS	4.56±0.98	254.2±33.4	1.72±1.19	79.1±42.4
RL-LNS	4.91±0.81	197.0±28.5	0.66±0.72	116.2±27.1
CL-LNS	**0.09±0.10**	**116.1±18.0**	**0.58±0.45**	**39.2±23.2**

Table 3.2: Primal gap (PG) (in percent), primal integral (PI) at 60-minute runtime cutoff, averaged over 100 test instances and their standard deviations. "↓" means the lower, the better. For ML methods, the policies are trained only on small training instances but are tested on both small and large test instances.

(a) MVC-S (left) and MVC-L (right).

(b) MIS-S (left) and MIS-L (right).

(c) CA-S (left) and CA-L (right).

(d) SC-S (left) and SC-L (right).
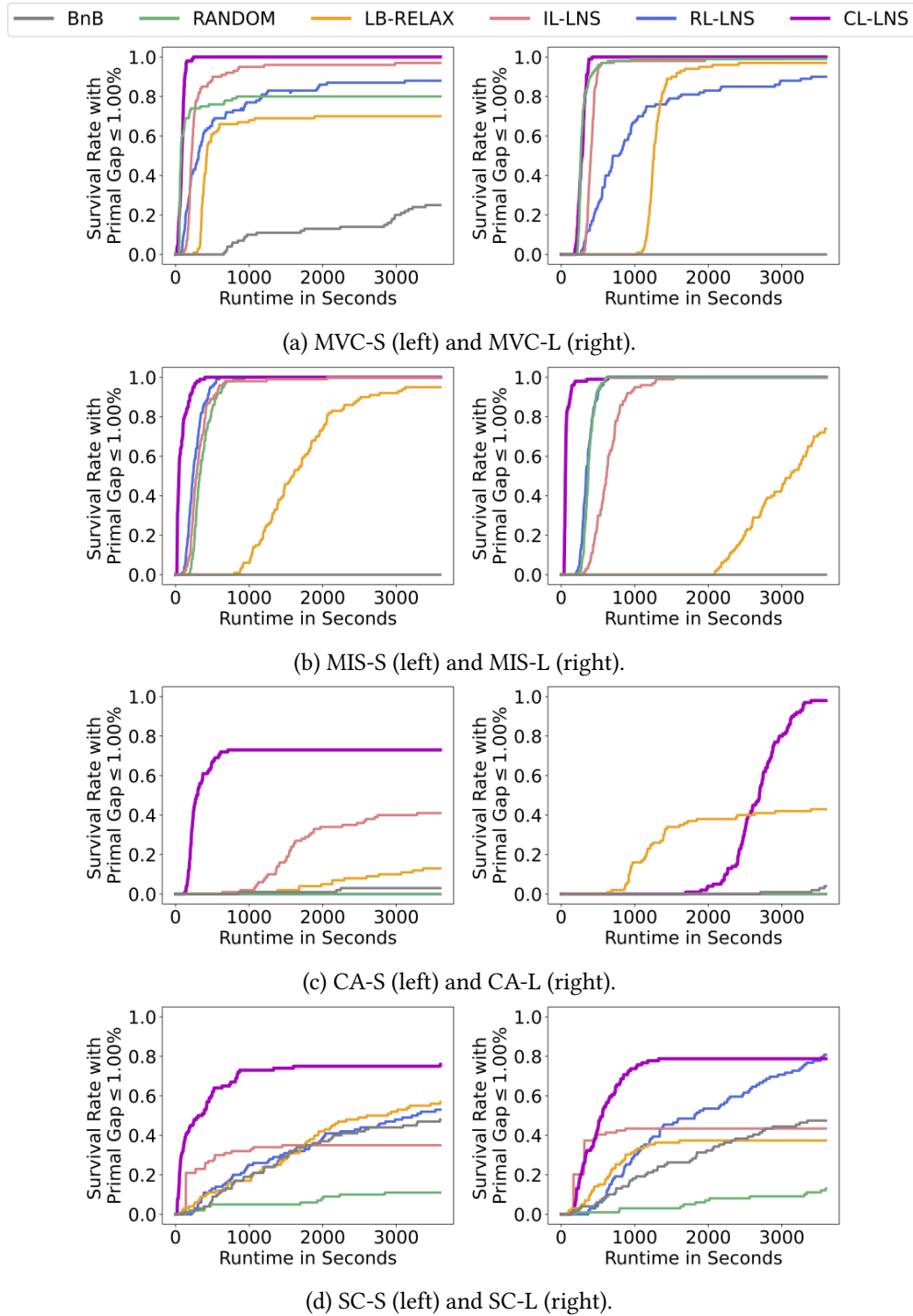
Figure 3.3: The survival rate (the higher, the better) over 100 test instances as a function of runtime to meet the primal gap threshold 1.00%. For ML methods, the policies are trained only on small training instances but are tested on both small and large test instances.

(a) MVC-S (left) and MVC-L (right).

(b) MIS-S (left) and MIS-L (right).

(c) CA-S (left) and CA-L (right).

(d) SC-S (left) and SC-L (right).
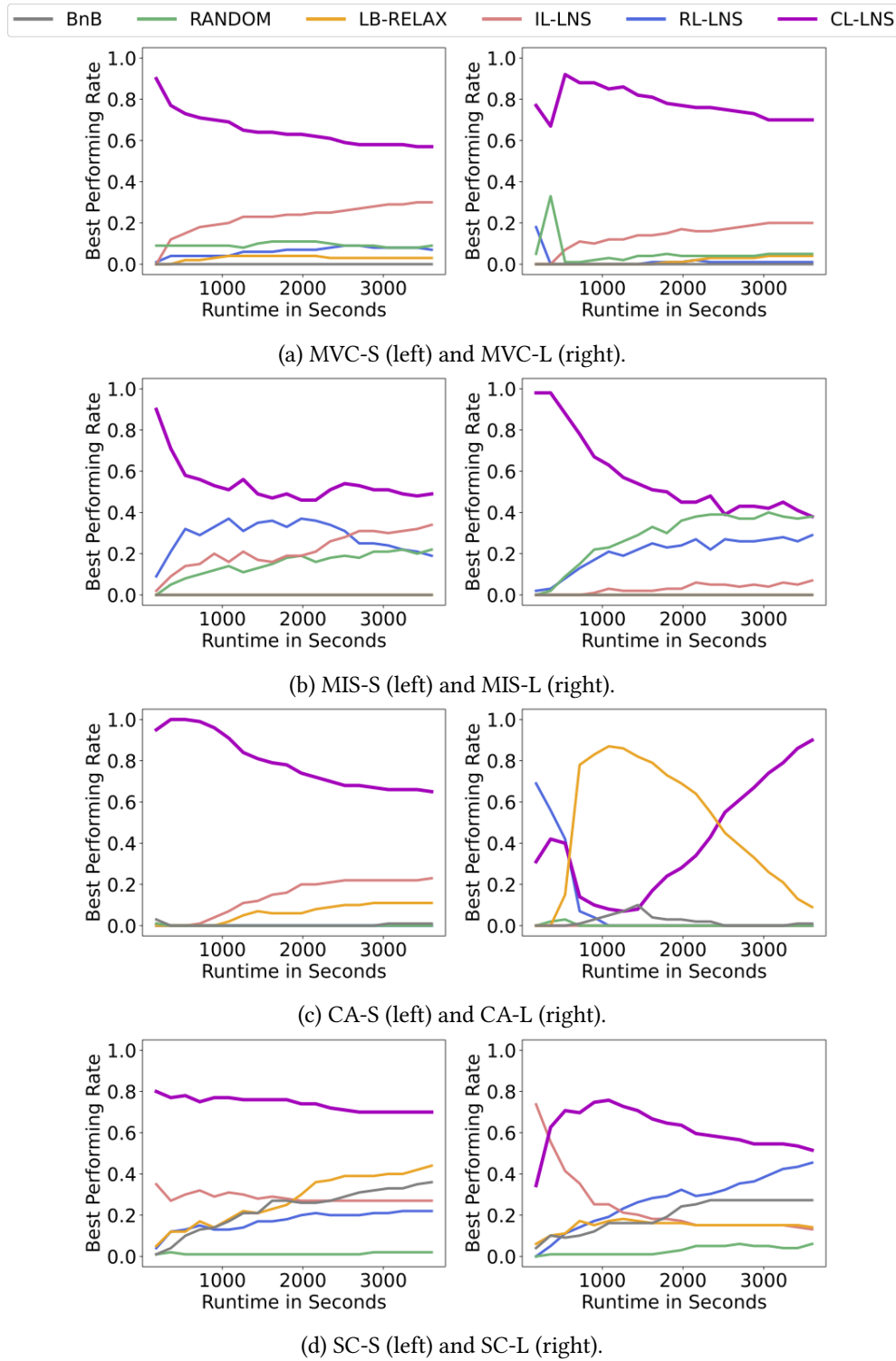
Figure 3.4: The best performing rate (the higher the better) as a function of runtime on 100 test instances. The sum of the best performing rates at a given runtime might sum up greater than 1 since ties are counted multiple times.

Comparison with LB (the Expert) Both IL-LNS and CL-LNS learn to imitate LB. On the small test instances, we run LB with two different neighborhood sizes, one that is fine-tuned in data collection and the other the same as CL-LNS, for 10 iterations and compare its per iteration performance with IL-LNS and CL-LNS. This allows us to compare the quality of the learned policies to the expert independently of their speed. The runtime limit per iteration for LB is set to 1 hour. Figure 3.5 shows the primal bound as a function of the number of iterations. The table in the figure summarizes the neighborhood sizes and the average runtime per iteration. For LB, the result shows that the neighborhood size affects the overall performance. Intuitively, using a larger neighborhood size in LB allows LNS to find better incumbent solutions by exploring larger neighborhoods. However, in practice, LB becomes less efficient in finding good incumbent solutions as the neighborhood size increases and sometimes even performs worse than using a smaller neighborhood size (the one for data collection). The neighborhood size for data collection is fine-tuned on validation instances to achieve the best primal bound upon convergences, allowing the ML models to observe demonstrations that lead to as good primal bounds as possible in training. However, when using the ML models in testing, we have the incentive to use a larger neighborhood size and fine-tune it since we no longer suffer from the bottleneck of LB. Therefore, we fine-tune the neighborhood sizes for IL-LNS and CL-LNS separately on validation instances. CL-LNS has a strong per-iteration performance that is consistently better than IL-LNS. With the fine-tuned neighborhood size, CL-LNS even outperforms the expert that it learns from (LB for data collection) on MIS-S and CA-S.

Ablation Study We evaluate how contrastive learning and two enhancements contribute to CL-LNS's performance. Compared to IL-LNS, CL-LNS uses (1) addition features from [102] and (2) GAT instead of GCN. We denote by "FF" the full feature set used in CL-LNS and "PF" the partial feature set in IL-LNS. In addition to IL-LNS and CL-LNS, we evaluate the performance of IL-LNS with FF and GAT (denoted by IL-LNS-GAT-FF), CL-LNS with GCN and PF (denoted by CL-LNS-GCN-PF) as well as CL-LNS with GAT and PF (denoted by CL-LNS-GAT-PF) on MVC-S and CA-S. Figure 3.6 shows the primal gap as a function

	MVC-S		MIS-S		CA-S		SC-S	
	NH size	Runtime	NH size	Runtime	NH size	Runtime	NH size	Runtime
LB	100	3600±0	3,000	3600±0	1,000	3600±0	100	3600±0
LB (data collection)	50	3600±0	500	3600±0	200	3600±0	50	3600±0
IL-LNS	100	2.1±0.1	3,000	1.3±0.2	1,000	20.8±13.1	150	120.9±1.3
CL-LNS	100	2.2±0.1	3,000	1.3±0.1	1,000	25.1±15.3	100	50.1±10.4
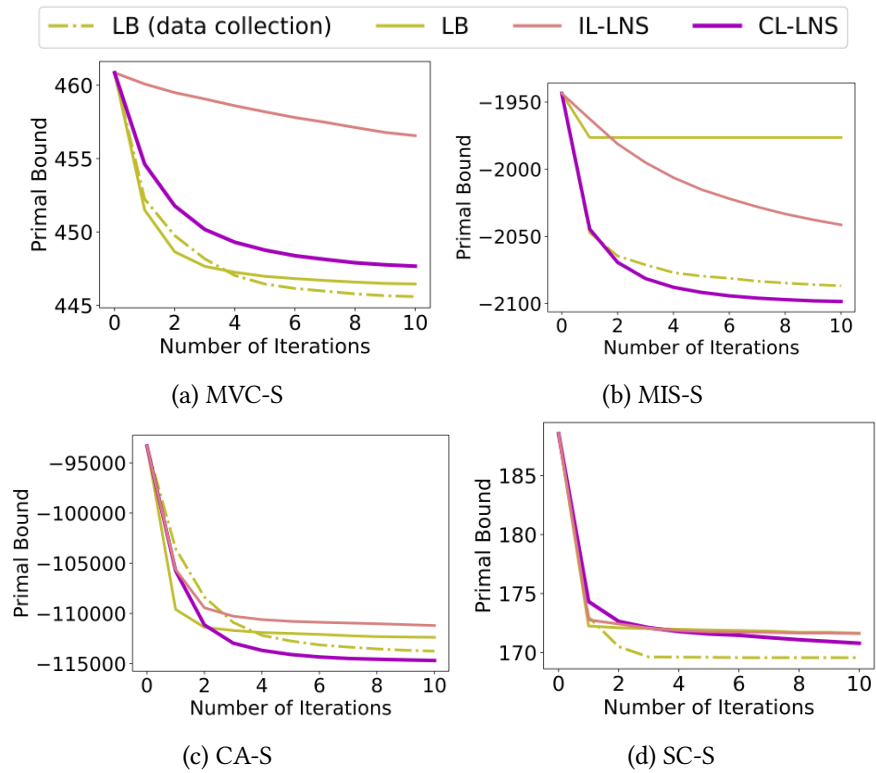
Figure 3.5: The primal bound (the lower, the better) as a function of the number of iterations averaged over 100 small test instances. LB and LB (data collection) are LNS with LB using the neighborhood sizes fine-tuned for CL-LNS and data collection, respectively. The table shows the neighborhood size (NH size) and the average runtime in seconds (with standard deviations) per iteration.

(a) MVC-S   (b) CA-S
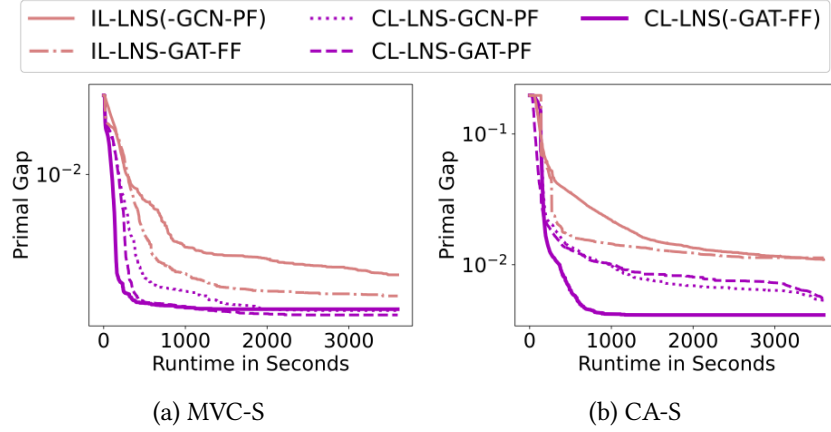
Figure 3.6: Ablation study: The primal gap (the lower, the better) as a function of time averaged over 100 small test instances.

	PG (%) ↓	PI ↓	PG (%) ↓	PI ↓
	MVC-S		CA-S	
IL-LNS(-GCN-PF)	0.29±0.23	19.2±10.2	1.09±0.51	90.0±20.8
IL-LNS-GAT-FF	0.24±0.17	15.3±7.3	1.13±0.63	78.9±22.7
CL-LNS-GCN-PF	0.17±0.10	11.4±8,8	0.75±0.40	57.9±21.2
CL-LNS-GAT-PF	**0.16±0.09**	10.1±0.6	0.76±0.39	53.8±22.1
CL-LNS(-GAT-FF)	0.17±0.09	**8.7±6.7**	**0.65±0.32**	**50.7±22.7**

Table 3.3: Ablation study: Primal gap (PG) (in percent) and primal integral (PI) at 60-minute runtime cutoff, averaged over 100 small test instances and their standard deviations. "↓" means the lower the better.

of runtime. Table 3.3 presents the primal gap and primal integral at a 60-minute runtime cutoff. The result

shows that IL-LNS-GAT-FF, imitation learning with the two enhancements, still performs worse than CL-

LNS-GCN-PF without any enhancements. CL-LNS-GCN-PF and CL-LNS-GAT-PF perform similarly in

terms of the primal gaps, but CL-LNS-GAT-PF has better primal integrals, showing the benefit of replacing

GCN with GAT. On MVC-S, CL-LNS and its other two variants have similar average primal gaps. On

CA-S, CL-LNS has a better average primal gap than the other two variants. However, adding the two

enhancements helps improve the primal integral, leading to the overall best performance of CL-LNS on

both MVC-S and CA-S.

3.7 Contrastive Predict-and-Search

In this section, we introduce *ConPaS*, **Con**trastive **P**redict-**a**nd-**S**earch for MILPs, to show how the framework can be applied to improve the predictions of partial assignments of values to variables in PaS. ConPaS leverages CL in the important task of learning to construct high-quality (partial) solutions to MILPs. A key to adapting the framework to this task is devising an appropriate and effective way of collecting positive and negative samples in this new context. Similar to both ND [148] and PaS [70], we collect a set of optimal and near-optimal solutions as *positive samples*; but different from ND and PaS, we additionally collect negative samples for CL. We propose to collect two types of negative samples - infeasible solutions and low-quality solutions that are similar to the positive samples - with novel approaches tailored to our task. For infeasible solutions, we use a sampling approach that randomly perturbs a small fraction of the positive samples. For low-quality solutions, we formulate the task as a maximin optimization. During training, instead of using a binary cross entropy loss to penalize the inaccurate predictions for each variable separately, we use a contrastive loss that encourages the model to predict solutions that are similar to the positive samples but dissimilar to the negative ones. Empirically, we test ConPaS on a variety of MILP problems, including problems from the NeurIPS Machine Learning for Combinatorial Optimization competition [58]. We show that ConPaS achieves state-of-the-art anytime performance on finding high-quality solutions to MILPs, substantially outperforming other learning-based methods such as ND and PaS in terms of solution quality and speed. In addition, ConPaS shows great generalization performance on test instances that are 50% larger than the training instances.

3.7.1 Machine Learning Methodology

For a given MILP M, our goal is to use CL to predict the conditional distribution of the solution $p(\boldsymbol{x}|M)$, such that it leads to high-quality solutions fast when it is used to guide downstream MILP solving. In this chapter, we mainly focus on using the prediction in Predict-and-Search (optimization problems (3.2))
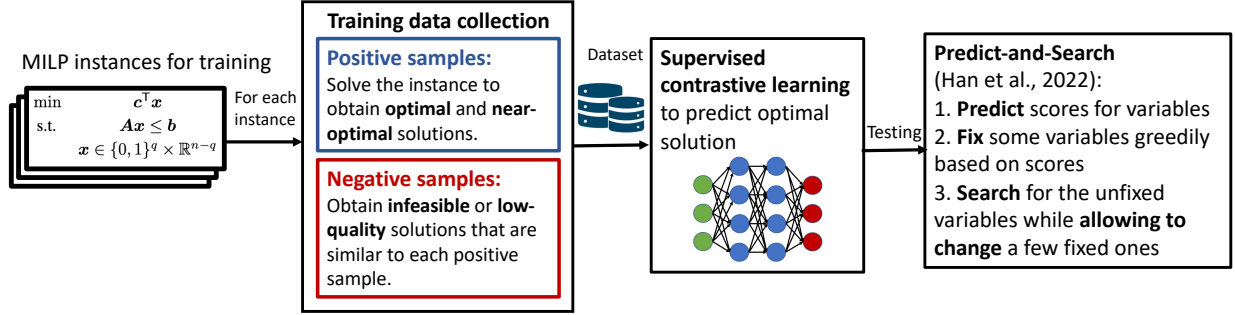
Figure 3.7: Overview of ConPaS. For training, we collect data from a set of MILP instances, including positive samples that are optimal and near-optimal solutions and negative samples that are low-quality or infeasible solutions. We use the data in supervised CL to predict optimal solutions. During testing, the predictions are used in Predict-and-Search [70].

following [70]. However, such prediction can be used to decompose the feasible regions of the input MILP for exact solving [44] or seed LNS with a better primal solution for heuristic solving [179]. Figure 3.7 gives an overview of ConPaS. Next, we describe our novel data collection, our supervised CL and how we apply solution predictions in the search.

3.7.1.1 Data Collection

In ConPaS, we use CL to learn to make discriminative predictions of optimal solutions by contrasting positive and negative samples. Since finding good assignments for integer variables is essentially the most challenging part of solving a MILP, we follow previous work [148] to learn $p(\boldsymbol{x}|M)$ approximately as $\prod_{i \leq q} p_\theta(x_i|M)$ where we mainly focus on predicting $p_\theta(x_i|M)$ for binary variables ($i \leq q$). Therefore, our definition of positive and negative samples of solutions mainly concerns the partial solutions on binary variables (since the optimal solutions for continuous variables can be computed in polynomial time once the binary ones are fixed). Now, we describe how we collect positive and negative samples.

Positive Samples Collection For a given MILP M, we collect a set of optimal or near-optimal solutions $\mathcal{S}_{\mathsf{p}}^M$ as our positive samples following previous works [148, 70]. This is done by solving M exhaustively

with a MILP solver and collecting up to u_p solutions with the minimum objective values. In experiments, u_p is set to 50.

Negative Samples Collection   Negative samples are critical parts of CL to help distinguish between high-quality and low-quality (or even infeasible) solutions. We propose to collect negative samples that are similar to the positive ones. From a theoretical point of view, the InfoNCE loss [151, 74] we use for training later can automatically focus on hard negative pairs (i.e., samples with similar representation but of very different qualities) and learn representations to separate them apart [187].

Given a MILP M, we collect a set of u_n negative samples $\mathcal{S}_\mathsf{n}^M$ where $u_\mathsf{n} = \beta_\mathsf{n}|\mathcal{S}_\mathsf{p}^M|$ and β_n is a hyperparameter to control the ratio between the number of positive and negative samples. In experiments, β_n is set to 10. We propose two novel approaches to collect them: (1) a sampling approach to collect infeasible solutions and (2) an optimization-based approach to collect low-quality solutions.

- **Infeasible Solutions as Negative Samples** We introduce a sampling approach. For each positive sample $x \in \mathcal{S}_\mathsf{p}^M$, we collect β_n infeasible solutions as negative samples. We randomly perturb 10% of the binary variable values in x (i.e., flipping from 0 to 1 or 1 to 0). If the MILP M contains only binary variables, we validate that the perturbed solutions are indeed infeasible if they violate at least one constraint in M. If M contains both binary and continuous variables, we fix the binary variables to the values in the perturbed solutions and ensure that no feasible assignment of the continuous variables exists using a MILP solver. If less than β_n negative samples are found after validating $2\beta_\mathsf{n}$ perturbed samples, we increase the perturbation rate by 5% and repeat the same process until we have β_n samples.

- **Low-Quality Solutions as Negative Samples** We introduce an optimization-based approach. For each positive sample $x = (x_1, \ldots, x_n) \in \mathcal{S}_\mathsf{p}^M$, we find the worst β_n feasible solutions that differ from

$\boldsymbol{x}$ in at most 10% of the binary variables. If the MILP $M = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, q)$ contains only binary variables, we find negative samples $\boldsymbol{x}'$ by solving the following Local Branching [56] MILP:

$$\max \boldsymbol{c}^\mathsf{T} \boldsymbol{x}'$$

$$\text{s.t.} \qquad \boldsymbol{A}\boldsymbol{x}' \leq \boldsymbol{b}, \, \boldsymbol{x}' \in \{0,1\}^q \times \mathbb{R}^{n-q}, \tag{3.3}$$

$$\textstyle\sum_{i \leq q : x_i = 0} x_i' + \sum_{i \leq q : x_i = 1} (1 - x_i') \leq k.$$

The above MILP is essentially solving the same problem as M, but with a negated objective function that tries to find solution x' as low-quality as possible and a constraint that allows changing at most k of the binary variables. After solving it, we consider only solutions as negative samples if they are worse than a given threshold. k is initially set to $10\% \times q$, but if less than β_n negative samples are found with the current k, we increase it by 5% and resolve optimization problem (3.3). We repeat the same process until we have β_n negative samples.

If M contains continuous variables, the goal is to find partial solutions on binary variables, such that we get as low-quality solutions $\boldsymbol{x}'$ as possible when we fix the binary values and optimize for the rest of the continuous variables. Formally, solving for the partial solutions on binary variables $x_1', \ldots, x_q'$ can be written as a maximin optimization:

$$\max_{x_1', \ldots, x_q'} \min_{x_{q+1}', \ldots, x_n'} \boldsymbol{c}^\mathsf{T} \boldsymbol{x}'$$

$$\text{s.t.} \qquad \boldsymbol{A}\boldsymbol{x}' \leq \boldsymbol{b}, \, \boldsymbol{x}' \in \{0,1\}^q \times \mathbb{R}^{n-q}, \tag{3.4}$$

$$\textstyle\sum_{i \leq q : x_i = 0} x_i' + \sum_{i \leq q : x_i = 1} (1 - x_i') \leq k.$$

Solving the above maximin optimization exactly is prohibitively hard and, to the best of our knowl-edge, there are no general-purpose solvers for it [11, Chapter 7]. Therefore, we use a heuristic ap-proach where we iteratively solve the inner minimization problem and add a constraint $\boldsymbol{c}^\mathsf{T}\boldsymbol{x}' > \boldsymbol{c}^\mathsf{T}\boldsymbol{x}^*$ to enforce the next solution found is strictly better than the current best-found solution $\boldsymbol{x}^*$ to the maximin problem. It terminates until no better solution can be found. For faster convergence, we sometimes enforce the next solution found to be at least $\epsilon > 0$ better than $\boldsymbol{x}^*$, i.e., we add $\boldsymbol{c}^\mathsf{T}\boldsymbol{x}' \geq \boldsymbol{c}^\mathsf{T}\boldsymbol{x}^* + \epsilon$, where ϵ is a hyperparameter tuned adaptively in a binary search manner. If we find less than β_n samples, we adjust k the same way as in the previous case.

3.7.1.2 Neural Network Architecture

Following previous work [70], we use a bipartite graph to represent the input MILP M. The bipartite graph has n variables and m constraints on two sides, respectively, with an edge connecting a variable and a constraint if the variable has a non-zero coefficient in the constraint. Following [148] and [70], we use node and edge features in the bipartite graph proposed by [59] . We learn $\boldsymbol{p_\theta}(\boldsymbol{x}|M)$ represented by a graph convolutional network (GCN) parameterized by learnable weights $\boldsymbol{\theta}$. The GCN takes the bipartite graph representation of M and the features as input. We perform two rounds of message passing through the GCN to obtain an embedding of the variables, which is then passed through a multi-layer perceptron (MLP) followed by a sigmoid activation layer to obtain the final output $p_{\boldsymbol{\theta}}(x_i|M)$. Details of the GCN architecture are included in Appendix.

3.7.1.3 Model Learning with a Contrastive Loss

Given a set of MILP instances $\mathcal{M}$ for training, let $\mathcal{D}^{\mathrm{ConPaS}} = \{(\mathcal{S}_\mathsf{p}^M, \mathcal{S}_\mathsf{n}^M) : M \in \mathcal{M}\}$ be the set of positive and negative samples for all training instances. A contrastive loss is a function whose value is low when the predicted $\boldsymbol{p_\theta}(\boldsymbol{x}|M)$ is similar to the positive samples $\mathcal{S}_\mathsf{p}^M$ and dissimilar to the negative samples $\mathcal{S}_\mathsf{n}^M$.

MILP Problem	MVC	MIS	CA	IP
#Binary Variables	6,000	6,000	4,000	1,050
#Continuous Variables	0	0	0	33
#Constraints	29,975	29,975	2,675	195

Table 3.4: The average numbers of variables and constraints in the test instances.

With similarity measured by dot products, we use an alternative form of InfoNCE, a supervised contrastive loss, that takes into account the solution qualities of both positive and negative samples:

$$\mathcal{L}^{\text{ConPaS}}(\boldsymbol{\theta}) = \sum_{(\mathcal{S}_{\mathsf{p}}^M, \mathcal{S}_{\mathsf{n}}^M) \in \mathcal{D}^{\text{ConPaS}}} \frac{-1}{|\mathcal{S}_{\mathsf{p}}^M|} \sum_{\boldsymbol{x}_{\mathsf{p}} \in \mathcal{S}_{\mathsf{p}}^M} \log \frac{\exp(\boldsymbol{x}_{\mathsf{p}}^\mathsf{T} \boldsymbol{p}_{\boldsymbol{\theta}}(\boldsymbol{x}|M)/\tau(\boldsymbol{x}_{\mathsf{p}}|M))}{\sum_{\boldsymbol{x}' \in \mathcal{S}_{\mathsf{n}}^M \cup \{\boldsymbol{x}_{\mathsf{p}}\}} \exp(\boldsymbol{x}'^\mathsf{T} \boldsymbol{p}_{\boldsymbol{\theta}}(\boldsymbol{x}|M)/\tau(\boldsymbol{x}'|M))}$$

where we let $\frac{1}{\tau(\boldsymbol{x}|M)} \propto -E(\boldsymbol{x}|M)$ if $\boldsymbol{x}$ is feasible to M where $E(\boldsymbol{x}|M)$ is the same energy function used in previous works [70, 148]; otherwise $\tau(\boldsymbol{x}|M)$ is set to a constant τ' ($\tau' = 1$ in experiments). Intuitively, setting $\tau(\boldsymbol{x}|M)$ in this manner encourages the predictions $\boldsymbol{p}_{\boldsymbol{\theta}}(\boldsymbol{x}|M)$ to be more similar to positive samples $\boldsymbol{x}_{\mathsf{p}}$ with better objectives.

3.7.1.4 ML-Guided Search

We apply the predicted solution to reduce the search space of the input MILP the same way as Predict-and-Search [70]. We greedily select $\mathcal{X}_0$ and $\mathcal{X}_1$ based on the prediction and solve the optimization problem defined by Equation (3.2) given hyperparameters k_0, k_1 and Δ.

3.7.2 Empirical Evaluation

In this subsection, we demonstrate the efficiency of ConPaS through experiments. In the following, we introduce our evaluation setup and then present the results.

3.7.2.1 Setup

MILP Problems We evaluate on four NP-hard MILP problems that are widely used in existing studies [59, 70], which consist of two graph optimization problems, namely the minimum vertex cover (MVC) and maximum independent set (MIS) problems, and two non-graph optimization problems, namely the combinatorial auction (CA) and item placement (IP) problems. Both MVC and MIS instances are generated according to the Barabasi-Albert random graph model [3], with 6,000 nodes and an average degree of 5. CA instances are generated with 2,000 items and 4,000 bids according to the arbitrary relations in [115]. IP instances are provided by the NeurIPS Machine Learning for Combinatorial Optimization competition [58]. The workload appointment problem is another MILP problem from the competition. However, they are not challenging enough for the baselines and our method. Therefore, we exclude the results on the workload appointment problem from the main content and report them in Appendix. For each problem, we have 400 training instances, 100 validation instances and 100 test instances. For each set of test instances, Table 3.4 shows its average numbers of variables and constraints. More details of instance generation are included in Appendix.

Baselines We compare ConPaS with three baselines: (1) SCIP (v8.0.1) [21], the state-of-the-art open-source ILP solver. We allow restart and presolving with the aggressive mode turned on for primal heuristics to focus on improving the objective value; (2) ND [148]; and (3) Predict-and-Search (PaS) [70]. We have considered another version of PaS where we replace the neural network output with the LP relaxation solutions of the MILP. However, this method causes very high infeasibility rates when solving the optimization problem defined by Equation (3.2). We also compare ConPaS with Gurobi (v10.0.0) [69] and present the results in Appendix.
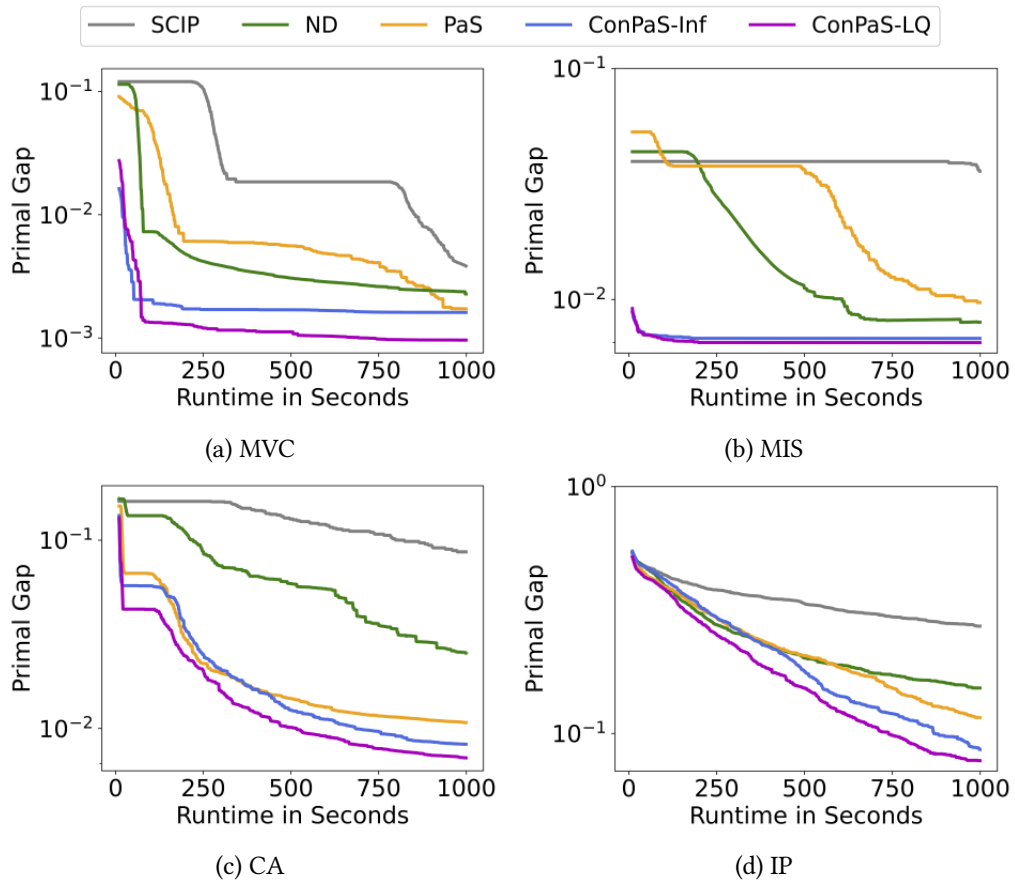
Figure 3.8: The primal gap (the lower the better) as a function of runtime, averaged over 100 test instances.

For ML-based methods, a separate model is trained for each MILP problem. For PaS, we train the models with the code provide by [70]. For ND, we implement it and fine-tune its hyperparameters for each MILP problem since their code is not available.

Metrics We use the following metrics to evaluate all methods: (1) The *primal gap* [19] is the normalized difference between the primal bound v and a precomputed best known objective value v^*, defined as $\frac{|v-v^*|}{\max(v,v^*,\varepsilon)}$ if v exists and $v \cdot v^* \geq 0$, or 1 otherwise. We use $\varepsilon = 10^{-8}$ to avoid division by zero; v^* is the best primal bound found within 60 minutes by any method in the portfolio for comparison; (2) The *primal integral* [1] at runtime cutoff t is the integral on $[0, t]$ of the primal gap as a function of runtime. It captures the quality of the solutions found and the speed at which they are found; and (3) The *survival rate* [179] to meet a certain primal gap threshold is the fraction of instances with primal gaps below the threshold.

Hyperparameters We conduct experiments on 2.4 GHz Intel Core i7 CPUs with 16 GB memory. Training is done on a NVIDIA P100 GPU with 32 GB memory. For data collection, we collect 50 best found solutions for each training instance with an hour runtime using Gurobi (v10.0.0). For training, we use the Adam optimizer [104] with learning rate 10^{-3}. We use a batch size of 8 and train for 100 epochs (the training typically converges in less than 50 epochs and 5 hours). For testing, we set the runtime cutoff to 1,000 seconds to solve the reduced MILP of each test instance with SCIP (v8.0.1).[*] To tune (k_0, k_1, Δ) (see definition in subsection 3.3.3) for both PaS and ConPaS, we first fix $\Delta = 5$ or 10 and vary k_0, k_1 to be $0\%, 10\%, \ldots, 50\%$ of the number of binary variables to test their performance on the validation instances to get their initial values. We then adjust Δ, k_0, k_1 around their initial values to find the best ones. The fine-tuned values are reported in Appendix.

---

[*]Note that our method is agnostic to the solver for the reduced MILP. The test results with Gurobi are reported in Appendix.
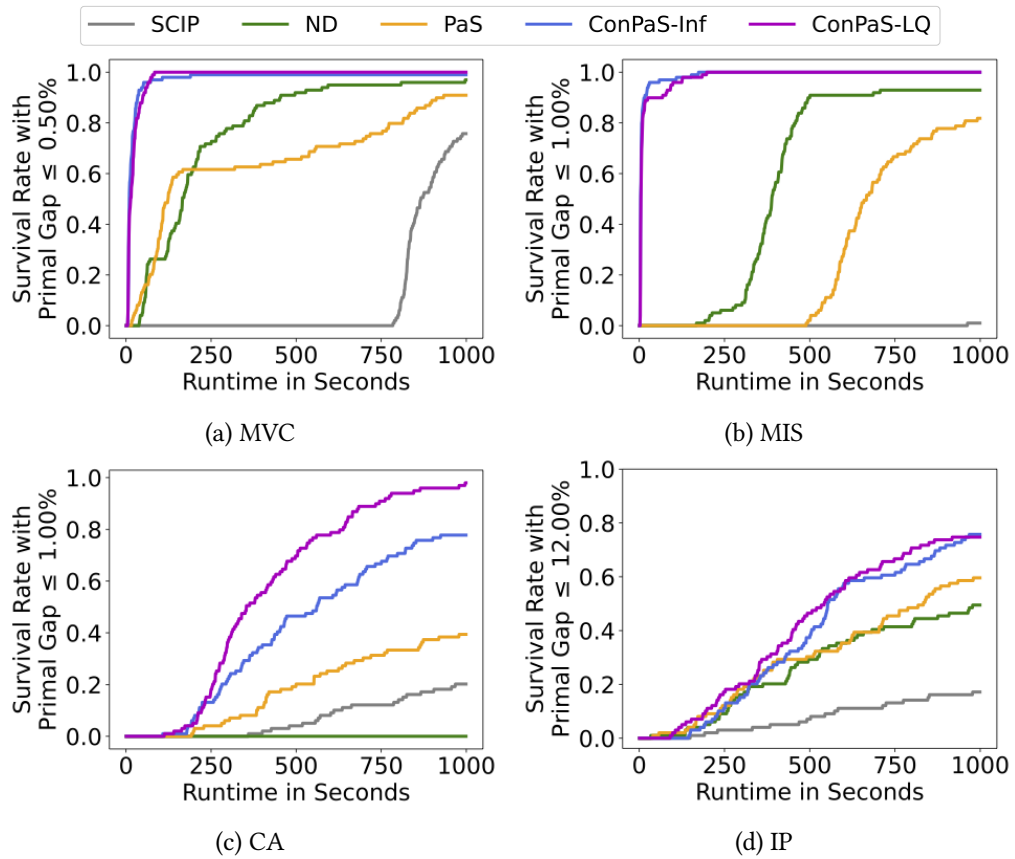
Figure 3.9: The survival rate (the higher, the better) to meet a certain primal gap threshold over 100 test instances as a function of runtime. The primal gap threshold is set to the median of the average primal gaps at the 1,000-second runtime cutoff among all methods rounded to the nearest 0.50%.

(a) MVC

(b) MIS

(c) CA

(d) IP
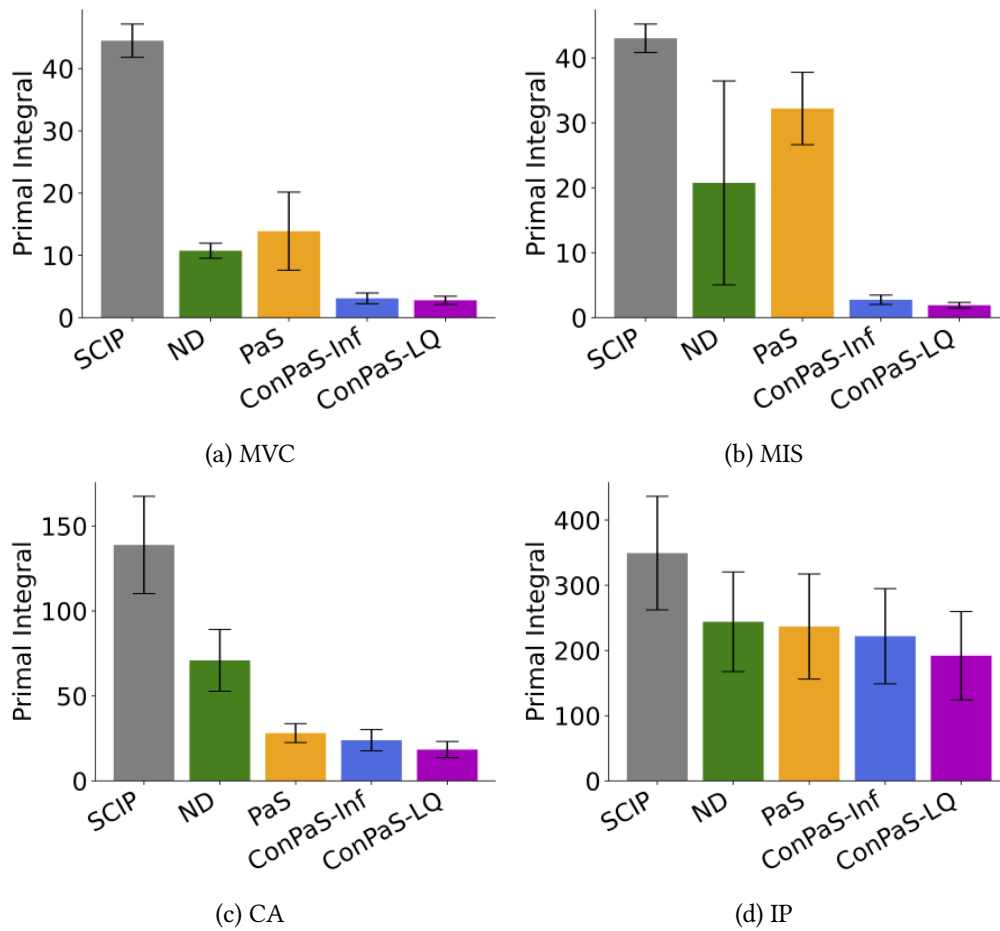
Figure 3.10: The primal integral (the lower, the better) at the 1,000-second runtime cutoff, averaged over 100 test instances. The error bars represent the standard deviation. A tabular representation is provided in the Appendix Table A.7.

(a) MVC (large instances).

(b) CA (large instances).
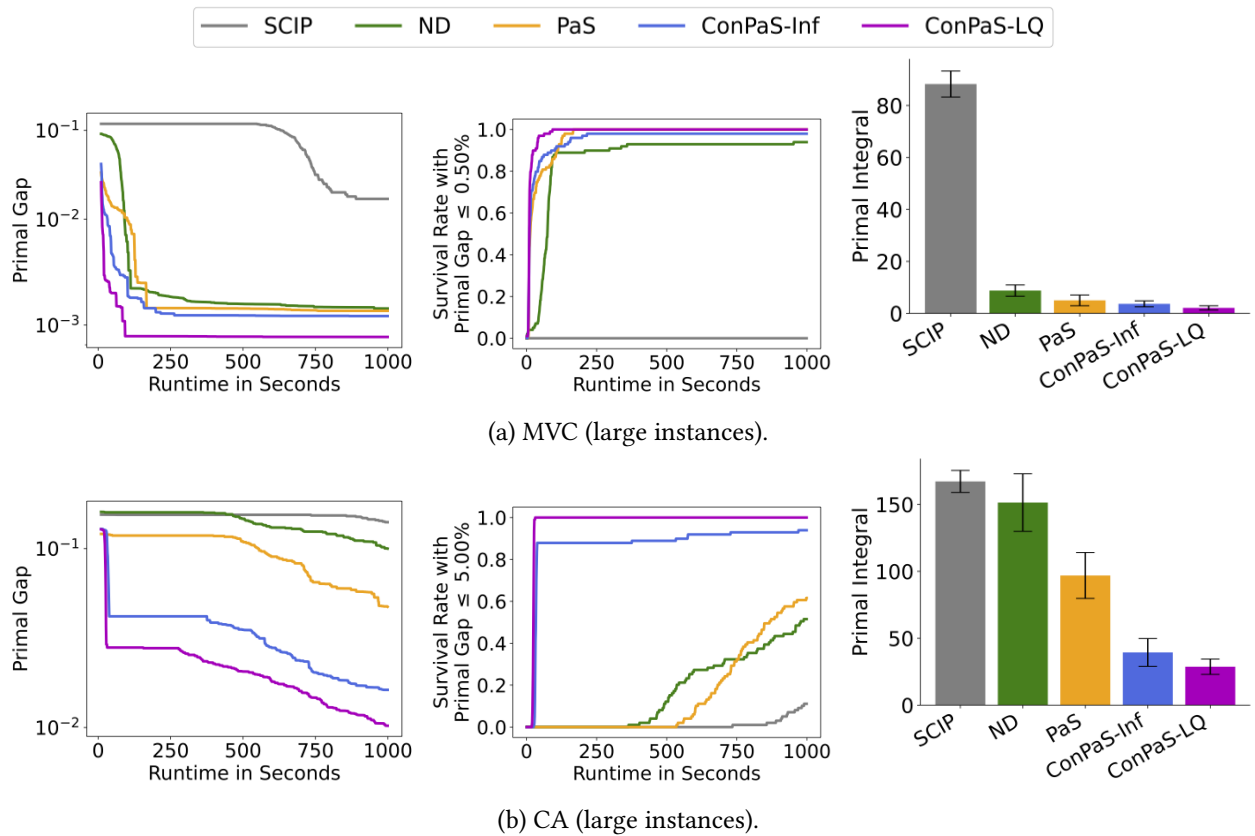
Figure 3.11: Generalization to 100 large instances: The primal gap as a function of runtime, the survival rate as a function of runtime and the primal integral at the 1,000-second runtime cutoff. The primal gap threshold for the survival rate is chosen as the medium of the average primal gaps at the 1,000-second runtime cutoff among all methods rounded to the nearest 0.50%. A tabular representation for the primal integral plots is provided in Appendix.
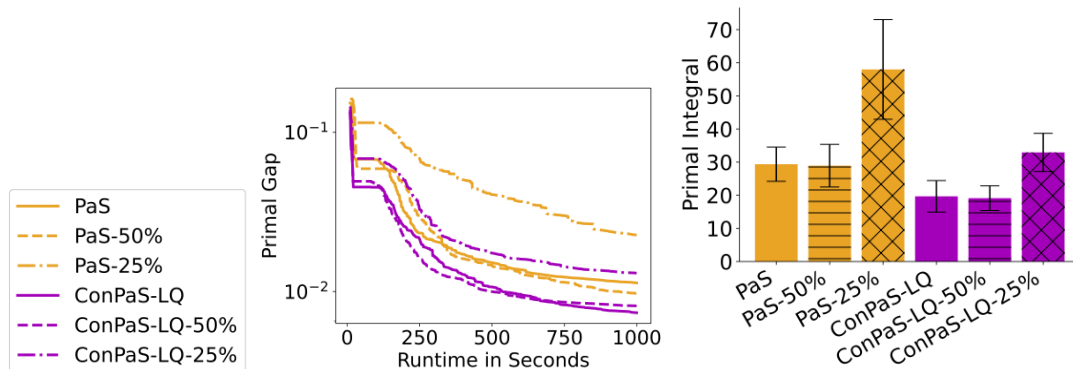
Figure 3.12: Training on different fractions of training instances: The primal gap as a function of runtime and the primal integral at the 1,000-second runtime cutoff. ConPaS-LQ-50% and ConPaS-LQ-25% denote the versions of ConPaS trained with only 50% and 25% of the training instances, respectively (similarly for PaS).

3.7.2.2 Results

We test two variants of ConPaS, denoted by ConPaS-Inf and ConPaS-LQ, that use infeasible solutions and low-quality solutions as negative samples, respectively. Figure 3.8 shows the primal gap as a function of runtime. Overall, SCIP performs the worst. PaS achieves lower average primal gaps than ND on three of the MILP problems at the 1,000-second runtime cutoff. Both ConPaS-Inf and ConPaS-LQ show better anytime performance than all baselines on all MILP problems. ConPaS-LQ performances slightly better than ConPaS-Inf. At the 1,000-second runtime cutoff, ConPaS-Inf achieves 3.54%-52.83% lower average primal gaps and ConPaS-LQ achieves 9.82%-86.02% lower average primal gaps than the best baseline.

Figure 3.9 shows the survival rate to meet a certain primal gap threshold. The primal gap threshold is chosen as the medium of the average primal gap at the 1,000-second runtime cutoff among all methods rounded to the nearest 0.50%. ND surprisingly has the lowest survival rate (even lower than SCIP) on the CA instances, indicating high variance in performance of both SCIP and ND[†], but ND is better than both SCIP and PaS on both the two graph optimization problems. PaS has higher survival rates on the CA and IP instances. ConPaS-Inf and ConPaS-LQ have the best survival rate at the 1,000-second runtime cutoff on all instances. Specifically, on the MVC and MIS instances, at the runtime cutoffs when they both first reach 100% survival rates, the best baseline only achieves about 10%-80% survival rates. These results indicate that ConPaS not only finds better solutions on average but also finds them on more instances. Figure 3.10 shows the average primal integral at the 1,000-second runtime cutoff. The result demonstrates that both ConPaS-Inf and ConPaS-LQ not only find better solutions than the other methods but also find them at a faster speed.

Next, we test the generalization performance and conduct an ablation study on the loss functions. Given the large computation overhead, we focus on two representative MILP problems, a graph optimization problem MVC and a non-graph optimization problem CA.

[†]When the primal gap threshold is set to 5.00%, ND has a 98% survival rate whereas SCIP has only 56%.

| | MVC | | CA | |
	PG	PI	PG	PI
PaS	0.17%	13.9	1.16%	28.9
ConPaS-LQ-unweighted	0.12%	3.3	0.57%	24.3
ConPaS-LQ	**0.10%**	**2.9**	**0.16%**	**19.7**

Table 3.5: Comparison of different loss functions. We report the primal gaps (PG) and the primal integrals (PI) at the 1,000-second runtime cutoff averaged over 100 instances.

Generalization to Larger Instances   We test the generalization performance of the trained models on larger instances. We generate 100 large MVC instances according to the Barabasi-Albert random graph model [3], with 9,000 nodes and an average degree of 5. We also generate 100 large CA instances with 3,000 items and 6,000 bids according to the arbitrary relations in [115]. These larger instances have 50% more variables and constraints than the previous test instances. In Figure 3.11, we show the results of the average primal gaps, survival rates and the average primal integral over 100 test instances. All ML-based methods demonstrate good generalizability. On large MVC instances, ND, PaS and ConPaS-Inf perform similarly in terms of the primal gap, while ConPaS-Inf improves the primal gap faster than the other methods. On large CA instances, both ConPaS-Inf and ConPaS-LQ are substantially better than the other baselines in terms of all performance metrics. Overall, on both large MVC and CA instances, ConPaS-LQ is the best and its primal integral at the 1,000-second runtime cutoff is 57.9%-70.3% lower than the best baseline PaS. It also reaches 100% survival rates fastest for the given thresholds.

Ablation Study   We conduct an ablation study on ConPaS-LQ to assess the effectiveness of the alternate form of InfoNCE loss. The results are shown in Table 3.5, where ConPaS-LQ-unweighted refers to training using the original InfoNCE loss without considering different qualities of the samples where we fine-tune and set $\tau(\boldsymbol{x}|M)$ to constant 1. ConPaS-LQ refers to the one that takes into account the solution qualities. ConPaS-LQ is still able to outperform PaS. Its performance further improves when the modified loss function is used.

	Primal Gap (%)		Primal Integral	
k_0	PaS	ConPaS-LQ	PaS	ConPaS-LQ
800	6.28	6.59	114.4	117.5
1200	5.45	5.05	104.3	97.3
1600	2.91	2.06	75.6	70.4
2000	1.17	**0.55**	28.9	**19.7**
2400	2.19	1.40	27.5	22.9
2700	5.63	4.58	58.0	47.4
3000	12.74	11.56	127.8	115.8

Table 3.6: The primal gap and primal integral at the 1,000-second runtime cutoff on the CA instances with different k_0 averaged over 100 instances.

The Effect of Hyperparameters We study the effect of hyperparameters. Specifically, we focus our study on PaS and ConPaS-LQ on the CA instances. We first empirically study how many training instances are needed for each method. We train separate models with 50% and 25% of the training instances and test their performance on the test instances. Figure 3.12 shows the results on the primal gap and primal integral. The two models for ConPaS-LQ trained with 50% and 100% of the instances perform similarly to each other. This is also true for PaS, but its two models are both worse than ConPaS-LQ. When we use 25% of the training instances, we observe a drop in performance for both methods. However, in this case, ConPaS-LQ performs much better than PaS and only slightly worse than PaS trained on 100% or 50% instances. These empirical results indicate that CL can achieve better performance using fewer training instances than other learning methods.

We also study the effect of different (k_0, k_1, Δ) for PaS and ConPaS-LQ on the CA instances. For CA instances, fixing both k_1 and Δ to 0 always gives better primal gaps and primal integrals than other values. Therefore, we vary only k_0. We present the results on primal gaps and primal integrals in Table 3.6. Overall, setting $k_0 = 2,000$ gives the best performance for both PaS and ConPaS-LQ. Either increasing or decreasing k_0 from 2,000 hurts their performance. However, if we increase k_0 from 2,000, both of them converge to the eventual solutions fast and therefore have comparable primal integrals with small k_0, even though sometimes their primal gaps are worse. In general, having a smaller k requires the search to search

for the values on more variables; therefore, it converges slower and has a larger primal integral. On the other hand, having a larger k reduces the search space more, therefore, it converges faster but to a worse solution.

3.8   Summary

In this chapter, we validated the hypothesis that one can leverage a general ML framework to improve human-designed decision-making strategies in different types of MILP search algorithms. We proposed a general ML framework based on contrastive learning. To apply this framework, we identify decisions in MILP search algorithms that we want to improve. Then, we collect training data for supervised CL. The training data includes positive and negative samples that are demonstrations of the decisions. Then, we train an ML model with a contrastive loss to predict decisions that are similar to the positive samples and dissimilar to negative ones. Finally, we use the learned ML model to make decisions during the search.

We first applied the framework to LNS and proposed CL-LNS that learned efficient and effective destroy heuristics in LNS. We presented a novel data collection process tailored for CL-LNS and used GAT with a richer set of features to further improve its performance. Empirically, CL-LNS substantially outperformed state-of-the-art methods on four MILP problems with respect to the primal gap, the primal integral, the best performing rate and the survival rate. CL-LNS achieved good generalization performance on out-of-distribution instances that are 100% larger than those used in training.

We then applied the framework to PaS and proposed ConPaS that learned to predict high-quality solutions by contrasting optimal and near-optimal solutions with infeasible or low-quality solutions. We presented a novel data collection process tailored for ConPaS, proposing a novel sampling-based approach and a novel optimization-based approach to collect negative samples. In testing, we solved a reduced-size MILP by restricting the search space to the proximity of the predicted solutions. Empirically, we showed

that ConPaS found solutions better and faster than the baselines, which include two state-of-the-art ML-guided MILP search algorithms. ConPaS achieved good generalization performance on out-of-distribution instances that are 50% larger than those used in training.

Chapter 4

Conclusions

In today's rapidly evolving society and economy, the scale, pace and variety of tasks related to resource allocation, design, planning and operations are expanding. These tasks are often subject to stringent resource constraints, high quality expectations and increasingly complex environments. Central to addressing the challenges of these tasks is addressing complex combinatorial optimization problems (COPs). In the past decades, search algorithms have been proposed to solve COPs. There are many decisions made by human-designed strategies in search algorithms that are crucial to their successful algorithmic advances. However, handcrafting those strategies is a complicated task that is prone to human errors and bias. On the other hand, machine learning (ML) has been the major force behind the successful advancements of many real-world applications nowadays. In this dissertation, we show that one can leverage general machine learning frameworks to improve human-designed decision-making strategies in different types of search algorithms for COPs. Specifically, we focus on two important COPs, namely multi-agent path finding (MAPF) and mixed integer linear programs (MILPs).

In Chapter 2, we presented our first major contributions to using ML to improving decision-making strategies in MAPF search algorithms. We contributed a general ML framework based on imitation learning and implemented the framework on four different MAPF search algorithms, namely CBS, ECBS, MAPF-LNS and PP, which substantially improved their performance in terms of runtime and/or solution quality.

The main contributions of this chapter were published in major artificial intelligence conferences individually in 2021 and 2022 [85, 89, 90, 213]. They are the first works that use ML techniques to enhance MAPF search algorithms by improving the quality of decision-making within the search process. Our works have inspired other works in the community since they were published [206, 192, 201, 156], where [206] uses a graph transformer architecture to improve node-selection strategies for ECBS, [192] uses genetic algorithms to learn priority-assignment strategies that can be expressed as arithmetic formulae for PP, [201] proposes a new deep neural network architecture to improve agent set-selection strategies for MAPF-LNS, and [156] uses online learning to learn to configure agent set-selection strategies. Both [192] and [201] are built upon our proposed ML framework. We believe that, by formulating a general ML framework and providing the four examples of implementations for different use cases, the contribution in Chapter 2 will serve as important guidance on how to improve MAPF search algorithms systematically.

Next, we discuss the limitations of the contribution in Chapter 2 and future work for improving decision-making strategies in MAPF search algorithms. The ML framework proposed in Chapter 2 uses imitation learning. Thus, one of its limitations is the need for computationally expensive data collection and an effective expert. Finding such an expert might require a good understanding of MAPF itself. For future work, it would be interesting to design unsupervised learning methods, such as reinforcement learning (RL) methods, to learn decision-making strategies in MAPF search algorithms without the need for data collection or an expert. RL has been applied to MAPF before but is mostly used to learn policies to construct conflict-free solutions for the agents [172, 38]. Applying RL to improving search algorithms poses a unique challenge since it is not straightforward to model some search algorithms, such as tree search, as a Markov decision process and the rewards are typically sparse, especially for difficult MAPF instances [173]. It would also be interesting future work to integrate deep learning techniques into the proposed ML framework. Though deep learning has been successfully used to improve solving MILPs and

other COPs [14, 212], it is challenging for MAPF due to the non-negligible computational overhead introduced by deep neural networks (DNNs) and the highly-optimized nature of state-of-the-art MAPF search algorithms. Recent work [201] has applied knowledge-distillation techniques to reduce the complexity of a DNN for MAPF-LNS. We believe designing engineering techniques to overcome such challenges is important and promising.

In Chapter 3, we presented our second major contribution to using ML to improve decision-making strategies in MILP search algorithms. We contributed a general ML framework based on contrastive learning (CL) and implemented the framework on two different MILP search algorithms, namely LNS and PaS, which substantially improved their performance in terms of both runtime and solution quality when compared to imitation learning and/or RL methods. The main contributions of this chapter were published in the International Conference on Machine Learning in 2023 and 2024 [87, 88]. Despite a lot of success in ML-guided MILP solving, they are the first works that apply CL techniques to improve MILP search algorithms. Our works have inspired other works in the ML-guided MILP-solving community since they were published [26, 55], where [26] apply the same CL framework to predict a subset of variables to prioritize branching on and [55, 106] use generative models to learn destroy heuristics for LNS. We believe that the contribution in Chapter 3 is valuable and will further facilitate the use of CL for ML-guided MILP solving.

Next, we discuss the limitations of the contribution in Chapter 3 and future work for improving decision-making strategies in MILP search algorithms. Solving MILPs based on solution predictions, such as ConPaS, does not guarantee completeness or optimality. CL-LNS does not either since it is based on LNS. Therefore, it would be interesting and important future work to integrate each of them into optimal MILP search algorithms such as Branch-and-Bound (BnB). For example, CL-LNS can be implemented as a primal heuristic, a class of heuristics that are capable of finding high-quality feasible solutions to the MILP fast in BnB. For this direction, it will be important to craft and utilize features related to the search tree of BnB, since BnB is a tree search that provides dynamic information, such as the dual bound of the

solution, cutting planes generated to prune the search space and branching decisions for partitioning the search space. On the other hand, ConPaS can also be incorporated in BnB, where the predicted solutions can be used to assign branching priorities to the variables or to select a linear combination of variables to branch on for generalized strong branching [202]. Furthermore, it is also promising future work to apply the CL framework to improve the performance of imitation learning or RL methods for decision-making in BnB, such as selecting variables to branch on and selecting nodes to expand.

To summarize, we presented two general ML frameworks to improve human-designed decision-making strategies in different search algorithms for MAPF and MILP, respectively. They are the first ML frameworks in the literature that provide guidance on how one could improve different search algorithms for a COP systematically. These frameworks are useful because one needs different search algorithms when the optimality requirements for the solutions and/or the computation budget to solve the COP change from time to time. Finally, we discuss how our contributions can be generalized to other COPs. First, it is a natural idea to apply the CL framework for MILP to MAPF. We discuss how this could be done for MAPF-LNS and PP and identify potential challenges. In Chapter 3, we showed that CL learned joint representations of variables given MILP instances that were useful for guiding decision-making for both LNS and PaS. For MAPF, we could leverage it to learn useful representations of agents given MAPF instances. For MAPF-LNS, instead of learning to select agent sets proposed by an expert in MAPF-ML-LNS, CL can be applied to learn to directly construct the agent sets, similarly to CL-LNS for MILPs. For PP, instead of predicting priorities for agents individually (even though we use features that capture information of other agents), we could leverage CL to learn to jointly predict a score for each agent to derive their priorities. To train the model, one could use scores or ranks for agents derived from good and bad total priority orderings as positive and negative samples, respectively. This approach would eliminate the need to assign labels using a heuristic method to account for multiple good total priority orderings, as we did for PP+ML. One important open question is to design features and ML model architectures for MAPF that capture dependencies

147

among agents, similar to the bipartite graph representations and graph neural networks for MILPs that capture dependencies between variables and constraints. Second, it is future work to apply either of the two ML frameworks to other COPs. Both frameworks used the same ML models, ML methods and loss functions as well as similar features for a COP, demonstrating good generalizability. We focused on training an ML model to make a single decision and evaluate that trained model only for making that decision. A promising way to apply the frameworks to COPs, in general, is to learn an ML model that generalizes across different COPs and/or decision-making tasks. In this direction, our ML frameworks can serve as the foundation for multi-task learning, where we can put together the data collected for different tasks with the frameworks to form a larger training dataset. We then use this larger training dataset to train a foundational ML model capable of performing various decision-making tasks in search algorithms for one or multiple COPs, with minimal or no fine-tuning required for each task.

Bibliography

[1]   Tobias Achterberg, Timo Berthold, and Gregor Hendel. "Rounding and propagation heuristics for mixed integer programming". In: *Operations Research Proceedings*. Springer, 2012, pp. 71–76.

[2]   Tobias Achterberg, Thorsten Koch, and Alexander Martin. "Branching rules revisited". In: *Operations Research Letters* 33.1 (2005), pp. 42–54.

[3]   Réka Albert and Albert-László Barabási. "Statistical mechanics of complex networks". In: *Reviews of modern physics* 74.1 (2002), p. 47.

[4]   André Altmann, Laura Toloşi, Oliver Sander, and Thomas Lengauer. "Permutation importance: a corrected feature importance measure". In: *Bioinformatics* 26.10 (2010), pp. 1340–1347.

[5]   Andre RS Amaral. "An exact approach to the one-dimensional facility layout problem". In: *Operations Research* 56.4 (2008), pp. 1026–1033.

[6]   Brandon Amos. "Tutorial on amortized optimization". In: *Foundations and Trends in Machine Learning* 16.5 (2023), pp. 592–732.

[7]   David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. *Finding cuts in the TSP (A preliminary report)*. Vol. 95. Citeseer, 1995.

[8]   Nabila Azi, Michel Gendreau, and Jean-Yves Potvin. "An adaptive large neighborhood search for a vehicle routing problem with multiple routes". In: *Computers & Operations Research* 41 (2014), pp. 167–173.

[9]   Jacopo Banfi, Nicola Basilico, and Francesco Amigoni. "Intractability of time-optimal multirobot path planning on 2d grid graphs with holes". In: *IEEE Robotics and Automation Letters* 2.4 (2017), pp. 1941–1947.

[10] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. "Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem". In: *Symposium on Combinatorial Search*. 2014, pp. 19–27.

[11] Yasmine Beck and Martin Schmidt. "A gentle and incomplete introduction to bilevel optimization". In: (2021). URL: https://optimization-online.org/?p=17182.

[12]   Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. "Neural combinatorial optimization with reinforcement learning". In: *arXiv preprint arXiv:1611.09940* (2016).

[13]   Stefano Benati and Romeo Rizzi. "A mixed integer linear programming formulation of the optimal mean/value-at-risk portfolio problem". In: *European Journal of Operational Research* 176.1 (2007), pp. 423–434.

[14]   Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. "Machine learning for combinatorial optimization: a methodological tour d'horizon". In: *European Journal of Operational Research* 290.2 (2021), pp. 405–421.

[15]   Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. "Curriculum learning". In: *International Conference on Machine Learning*. 2009, pp. 41–48.

[16]   Maren Bennewitz, Wolfram Burgard, and Sebastian Thrun. "Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots". In: *Robotics and Autonomous Systems* 41.2-3 (2002), pp. 89–99.

[17]   Jur P. van den Berg and Mark H. Overmars. "Prioritized motion planning for multiple robots". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2005, pp. 430–435.

[18]   Jur P. van den Berg, Jack Snoeyink, Ming C. Lin, and Dinesh Manocha. "Centralized path planning for multiple robots: Optimal decoupling into sequential plans". In: *Robotics: Science and Systems V*. 2009, pp. 2–3.

[19]   Timo Berthold. "Primal heuristics for mixed integer programs". PhD thesis. Zuse Institute Berlin (ZIB), 2006.

[20]   Timo Berthold. "RENS". In: *Mathematical Programming Computation* 6.1 (2014), pp. 33–54.

[21]   Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. *The SCIP optimization suite 8.0*. Technical Report. Optimization Online, Dec. 2021. URL: http://www.optimization-online.org/DB_HTML/2021/12/8728.html.

[22]   Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Eyal Shimony. "ICBS: Improved conflict-based search algorithm for multi-agent pathfinding". In: *International Joint Conference on Artificial Intelligence*. 2015, pp. 442–449.

[23]   Shaked Brody, Uri Alon, and Eran Yahav. "How attentive are graph attention networks?" In: *International Conference on Learning Representations* (2022).

[24]   Stephen J Buckley. "Fast motion planning for multiple moving robots". In: *IEEE International Conference on Robotics and Automation*. 1989, pp. 322–326.

[25]   Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. "Learning to rank using gradient descent". In: *International Conference on Machine Learning*. 2005, pp. 89–96.

[26]   Junyang Cai, Taoan Huang, and Bistra Dilkina. "Learning backdoors for mixed integer programs with contrastive learning". In: *arXiv preprint arXiv:2401.10467* (2024).

[27]   Yi Cao, Sivakumar Rathinam, and Dengfeng Sun. "Greedy-heuristic-aided mixed-integer linear programming approach for arrival scheduling". In: *Journal of Aerospace Information Systems* 10.7 (2013), pp. 323–336.

[28]   Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre A Cire. "Combining reinforcement learning and constraint programming for combinatorial optimization". In: *AAAI Conference on Artificial Intelligence*. Vol. 35. 5. 2021, pp. 3677–3687.

[29]   Gary W Chang, YD Tsai, CY Lai, and JS Chung. "A practical mixed integer linear programming based approach for unit commitment". In: *IEEE Power Engineering Society General Meeting, 2004*. IEEE. 2004, pp. 221–225.

[30]   Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. "A simple framework for contrastive learning of visual representations". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 1597–1607.

[31]   Weizhe Chen, Zhihan Wang, Jiaoyang Li, Sven Koenig, and Bistra Dilkina. "No panacea in planning: Algorithm selection for suboptimal multi-agent path finding". In: *arXiv preprint arXiv:2404.03554* (2024).

[32]   Xinyun Chen and Yuandong Tian. "Learning to perform local rewriting for combinatorial optimization". In: *Advances in Neural Information Processing Systems* 32 (2019).

[33]   Zhe Chen, Daniel Harabor, Jiaoyang Li, and Peter J Stuckey. "Traffic flow optimisation for lifelong multi-agent path finding". In: *AAAI Conference on Artificial Intelligence*. Vol. 38. 18. 2024, pp. 20674–20682.

[34]   Antonia Chmiela, Elias Khalil, Ambros Gleixner, Andrea Lodi, and Sebastian Pokutta. "Learning to schedule heuristics in branch and bound". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 24235–24246.

[35]   Liron Cohen, Tansel Uras, and Sven Koenig. "Feasibility study: Using highways for bounded-suboptimal multi-agent path finding". In: *Symposium on Combinatorial Search*. 2015.

[36]   Liron Cohen, Tansel Uras, TK Satish Kumar, Hong Xu, Nora Ayanian, and Sven Koenig. "Improved solvers for bounded-suboptimal multi-agent path finding." In: *International Joint Conference on Artificial Intelligence*. 2016, pp. 3067–3074.

[37] IBM ILOG Cplex. "V12. 1: User's manual for CPLEX". In: *International Business Machines Corporation* 46.53 (2009), p. 157.

[38] Mehul Damani, Zhiyao Luo, Emerson Wenzel, and Guillaume Sartoretti. "PRIMAL_2: Pathfinding via reinforcement and imitation multi-agent learning-lifelong". In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 2666–2673.

[39] Emilie Danna, Edward Rothberg, and Claude Le Pape. "Exploring relaxation induced neighborhoods to improve MIP solutions". In: *Mathematical Programming* 102.1 (2005), pp. 71–90.

[40] Hal Daumé, John Langford, and Daniel Marcu. "Search-based structured prediction". In: *Machine Learning* 75.3 (2009), pp. 297–325.

[41] Sven De Vries and Rakesh V Vohra. "Combinatorial auctions: A survey". In: *INFORMS Journal on computing* 15.3 (2003), pp. 284–309.

[42] Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. "Learning heuristics for the TSP by policy gradient". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2018, pp. 170–181.

[43] Bistra Dilkina and Carla P Gomes. "Solving connected subgraph problems in wildlife conservation." In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Vol. 6140. Springer. 2010, pp. 102–116.

[44] Jian-Ya Ding, Chao Zhang, Lei Shen, Shengyin Li, Bing Wang, Yinghui Xu, and Le Song. "Accelerating primal solution findings for mixed integer programs based on solution prediction". In: *AAAI Conference on Artificial Intelligence*. Vol. 34. 02. 2020, pp. 1452–1459.

[45] Kurt Dresner and Peter Stone. "A multiagent approach to autonomous intersection management". In: *Journal of Artificial Intelligence Research* 31 (2008), pp. 591–656.

[46] Haonan Duan, Pashootan Vaezipoor, Max B Paulus, Yangjun Ruan, and Chris Maddison. "Augment with care: Contrastive learning for combinatorial problems". In: *International Conference on Machine Learning*. PMLR. 2022, pp. 5627–5642.

[47] Lu Duan, Haoyuan Hu, Yu Qian, Yu Gong, Xiaodong Zhang, Jiangwen Wei, and Yinghui Xu. "A multi-task selected learning approach for solving 3D flexible bin packing problem". In: *International Conference on Autonomous Agents and MultiAgent Systems*. 2019, pp. 1386–1394.

[48] Olivier Dubois and Gilles Dequen. "A backbone-search heuristic for efficient solving of hard 3-SAT formulae". In: *International Joint Conference on Artificial Intelligence*. Vol. 1. 2001, pp. 248–253.

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

[50] Paul Erdos, Alfréd Rényi, et al. "On the evolution of random graphs". In: *Publ. Math. Inst. Hung. Acad. Sci* 5.1 (1960), pp. 17–60.

[51] Benjamin Eysenbach, Tianjun Zhang, Sergey Levine, and Russ R Salakhutdinov. "Contrastive learning as goal-conditioned reinforcement learning". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 35603–35620.

[52] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. "LIBLINEAR: A library for large linear classification". In: *Journal of Machine Learning Research* 9.Aug (2008), pp. 1871–1874.

[53] Ariel Felner, Meir Goldenberg, Guni Sharon, Roni Stern, Tal Beja, Nathan R Sturtevant, Jonathan Schaeffer, and Robert Holte. "Partial-Expansion A* with Selective Node Generation." In: *AAAI Conference on Artificial Intelligence.* 2012, pp. 180–181.

[54] Ariel Felner, Jiaoyang Li, Eli Boyarski, Hang Ma, Liron Cohen, TK Satish Kumar, and Sven Koenig. "Adding heuristics to conflict-based search for multi-agent path finding". In: *International Conference on Automated Planning and Scheduling.* 2018, pp. 83–87.

[55] Shengyu Feng, Zhiqing Sun, and Yiming Yang. "DIFUSCO-LNS: Diffusion-guided large neighborhood search for integer linear programming". In: (2023).

[56] Matteo Fischetti and Andrea Lodi. "Local branching". In: *Mathematical programming* 98.1 (2003), pp. 23–47.

[57] Graeme Gange, Daniel Harabor, and Peter J Stuckey. "Lazy CBS: implicit conflict-based search using lazy clause generation". In: *International Conference on Automated Planning and Scheduling.* Vol. 29. 2019, pp. 155–162.

[58] Maxime Gasse, Simon Bowly, Quentin Cappart, Charfreitag, et al. "The machine learning for combinatorial optimization competition (ml4co): Results and insights". In: *NeurIPS 2021 Competitions and Demonstrations Track.* PMLR. 2022, pp. 220–231.

[59] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. "Exact combinatorial optimization with graph convolutional neural networks". In: *Advances in Neural Information Processing Systems* 32 (2019).

[60] Yonatan Geifman and Ran El-Yaniv. "Selectivenet: A deep neural network with an integrated reject option". In: *International Conference on Machine Learning.* PMLR. 2019, pp. 2151–2159.

[61] Shubhashis Ghosh. "DINS, a MIP improvement heuristic". In: *International Conference on Integer Programming and Combinatorial Optimization.* Springer. 2007, pp. 310–323.

[62] John Giorgi, Osvald Nitski, Bo Wang, and Gary Bader. "DeCLUTR: Deep contrastive learning for unsupervised textual representations". In: *Annual Meeting of the Association for Computational Linguistics and International Joint Conference on Natural Language Processing (Volume 1: Long Papers).* 2021, pp. 879–895.

[63] Rodrigo N Gómez, Carlos Hernández, and Jorge A Baier. "A compact answer set programming encoding of multi-agent pathfinding". In: *IEEE Access* 9 (2021), pp. 26886–26901.

[64] Ralph E Gomory. *Outline of an algorithm for integer solutions to linear programs and an algorithm for the mixed integer problem.* Springer, 2010.

[65] Lacy M Greening, Mathieu Dahan, and Alan L Erera. "Lead-time-constrained middle-mile consolidation network design with fixed origins and destinations". In: *Transportation Research Part B: Methodological* 174 (2023), p. 102782.

[66] Beliz Gunel, Jingfei Du, Alexis Conneau, and Veselin Stoyanov. "Supervised contrastive learning for pre-trained language model fine-tuning". In: *International Conference on Learning Representations.* 2021.

[67] Amrita Gupta and Bistra Dilkina. "Budget-constrained demand-weighted network design for resilient infrastructure". In: *2019 IEEE 31st International Conference on Tools with Artificial Intelligence.* IEEE. 2019, pp. 456–463.

[68] Prateek Gupta, Maxime Gasse, Elias Khalil, Pawan Mudigonda, Andrea Lodi, and Yoshua Bengio. "Hybrid models for learning to branch". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 18087–18097.

[69] Gurobi Optimization, LLC. *Gurobi optimizer reference manual.* 2022. URL: https://www.gurobi.com.

[70] Qingyu Han, Linxin Yang, Qian Chen, et al. "A GNN-guided predict-and-search framework for mixed-integer linear programming". In: *International Conference on Learning Representations.* 2022.

[71] Pierre Hansen, Nenad Mladenović, Jack Brimberg, and José A Moreno Pérez. *Variable neighborhood search.* Springer, 2019.

[72] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[73] He He, Hal Daume III, and Jason M. Eisner. "Learning to search in branch and bound algorithms". In: *Advances in Neural Information Processing Systems.* 2014, pp. 3293–3301.

[74] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. "Momentum contrast for unsupervised visual representation learning". In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2020, pp. 9729–9738.

[75] Gregor Hendel. "Adaptive large neighborhood search for mixed integer programming". In: *Mathematical Programming Computation* 14.2 (2022), pp. 185–221.

[76] Sunderesh S Heragu and Andrew Kusiak. "Efficient models for the facility layout problem". In: *European Journal of Operational Research* 53.1 (1991), pp. 1–13.

[77] R Devon Hjelm, Alex Fedorov, Samuel Lavoie-Marchildon, Karan Grewal, Phil Bachman, Adam Trischler, and Yoshua Bengio. "Learning deep representations by mutual information estimation and maximization". In: *International Conference on Learning Representations* (2019).

[78]   Florence Ho, Rúben Geraldes, Artur Gonçalves, Bastien Rigault, Benjamin Sportich, Daisuke Kubo, Marc Cavazza, and Helmut Prendinger. "Decentralized multi-agent path finding for UAV traffic management". In: *IEEE Transactions on Intelligent Transportation Systems* 23.2 (2020), pp. 997–1008.

[79]   John H Holland. "Genetic algorithms". In: *Scientific American* 267.1 (1992), pp. 66–73.

[80]   Wolfgang Hönig, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. "Persistent and robust execution of MAPF schedules in warehouses". In: *IEEE Robotics and Automation Letters* 4.2 (2019), pp. 1125–1131.

[81]   Wolfgang Hönig, James A Preiss, TK Satish Kumar, Gaurav S Sukhatme, and Nora Ayanian. "Trajectory planning for quadrotor swarms". In: *IEEE Transactions on Robotics* 34.4 (2018), pp. 856–869.

[82]   André Hottung and Kevin Tierney. "Neural large neighborhood search for the capacitated vehicle routing problem". In: *European Conference on Artificial Intelligence*. IOS Press, 2020, pp. 443–450.

[83]   Haoyuan Hu, Xiaodong Zhang, Xiaowei Yan, Longfei Wang, and Yinghui Xu. "Solving a new 3d bin packing problem with deep reinforcement learning method". In: *arXiv preprint arXiv:1708.05930* (2017).

[84]   Taoan Huang and Bistra Dilkina. "Enhancing seismic resilience of water pipe networks". In: *ACM SIGCAS Conference on Computing and Sustainable Societies*. 2020, pp. 44–52.

[85]   Taoan Huang, Bistra Dilkina, and Sven Koenig. "Learning node-selection strategies in bounded suboptimal conflict-based search for multi-agent path finding". In: *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 2021.

[86]   Taoan Huang, Aaron Ferber, Yuandong Tian, Bistra Dilkina, and Benoit Steiner. "Local branching relaxation heuristics for integer linear programs". In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2023, pp. 96–113.

[87]   Taoan Huang, Aaron M Ferber, Yuandong Tian, Bistra Dilkina, and Benoit Steiner. "Searching large neighborhoods for integer linear programs with contrastive learning". In: *International Conference on Machine Learning*. PMLR. 2023, pp. 13869–13890.

[88]   Taoan Huang, Aaron M Ferber, Arman Zharmagambetov, Yuandong Tian, and Bistra Dilkina. "Contrastive predict-and-search for mixed integer linear programs". In: *International Conference on Machine Learning*. PMLR. 2024.

[89]   Taoan Huang, Sven Koenig, and Bistra Dilkina. "Learning to resolve conflicts for multi-agent path finding with conflict-based search". In: *AAAI Conference on Artificial Intelligence*. Vol. 35. 13. 2021, pp. 11246–11253.

[90]   Taoan Huang, Jiaoyang Li, Sven Koenig, and Bistra Dilkina. "Anytime multi-agent path finding via machine learning-guided large neighborhood search". In: *AAAI Conference on Artificial Intelligence*. Vol. 36. 9. 2022, pp. 9368–9376.

[91]   Taoan Huang, Vikas Shivashankar, Michael Caldara, Joseph Durham, Jiaoyang Li, Bistra Dilkina, and Sven Koenig. "Deadline-aware multi-agent tour planning". In: *International Conference on Automated Planning and Scheduling*. 2023.

[92]   Zeren Huang, Kerong Wang, Furui Liu, Hui-Ling Zhen, Weinan Zhang, Mingxuan Yuan, Jianye Hao, Yong Yu, and Jun Wang. "Learning to select cuts for efficient mixed-integer programming". In: *Pattern Recognition* 123 (2022), p. 108353.

[93]   Anil Jindal and Kuldip Singh Sangwan. "Closed loop supply chain network design and optimisation using fuzzy mixed integer linear programming model". In: *International Journal of Production Research* 52.14 (2014), pp. 4156–4173.

[94]   Thorsten Joachims. "Optimizing search engines using clickthrough data". In: *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2002, pp. 133–142.

[95]   Thorsten Joachims. "Training linear SVMs in linear time". In: *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2006, pp. 217–226.

[96]   David S Johnson, Jan Karel Lenstra, and AHG Rinnooy Kan. "The complexity of the network design problem". In: *Networks* 8.4 (1978), pp. 279–285.

[97]   David S Johnson, Christos H Papadimitriou, and Mihalis Yannakakis. "How easy is local search?" In: *Journal of computer and system sciences* 37.1 (1988), pp. 79–100.

[98]   Omri Kaduri, Eli Boyarski, and Roni Stern. "Algorithm selection for optimal multi-agent pathfinding". In: *International Conference on Automated Planning and Scheduling*. 2020, pp. 161–165.

[99]   Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. "Learning combinatorial optimization algorithms over graphs". In: *Advances in Neural Information Processing Systems* 30 (2017).

[100] Elias B Khalil, Bistra Dilkina, George L Nemhauser, Shabbir Ahmed, and Yufen Shao. "Learning to Run Heuristics in Tree Search." In: *International Joint Conference on Artificial Intelligence*. 2017, pp. 659–666.

[101] Elias B Khalil, Christopher Morris, and Andrea Lodi. "Mip-gnn: A data-driven framework for guiding combinatorial solvers". In: *AAAI Conference on Artificial Intelligence*. Vol. 36. 9. 2022, pp. 10219–10227.

[102] Elias Boutros Khalil, Pierre Le Bodic, Le Song, George L. Nemhauser, and Bistra Dilkina. "Learning to branch in mixed integer programming". In: *AAAI Conference on Artificial Intelligence*. 2016, pp. 724–731.

[103] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. "Supervised contrastive learning". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 18661–18673.

[104] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *International Conference on Learning Representations*. 2015.

[105] Satoshi Kitayama and Keiichiro Yasuda. "A method for mixed integer programming problems by particle swarm optimization". In: *Electrical Engineering in Japan* 157.2 (2006), pp. 40–49.

[106] Shufeng Kong, Caihua Liu, and Carla P Gomes. "ILP-FORMER: Solving Integer Linear Programming with Sequence to Multi-Label Learning". In: *Uncertainty in Artificial Intelligence*. 2024.

[107] Wouter Kool, Herke Van Hoof, and Max Welling. "Attention, learn to solve routing problems!" In: *International Conference on Learning Representations*. 2018.

[108] Attila A Kovacs, Sophie N Parragh, Karl F Doerner, and Richard F Hartl. "Adaptive large neighborhood search for service technician routing and scheduling problems". In: *Journal of Scheduling* 15.5 (2012), pp. 579–600.

[109] Wen-Yang Ku and J Christopher Beck. "Mixed integer programming models for job shop scheduling: A computational analysis". In: *Computers & Operations Research* 73 (2016), pp. 165–173.

[110] Abdel Ghani Labassi, Didier Chételat, and Andrea Lodi. "Learning to compare nodes in branch and bound with graph neural networks". In: *Advances in Neural Information Processing Systems* (2022).

[111] Edward Lam and Pierre Le Bodic. "New valid inequalities in branch-and-cut-and-price for multi-agent path finding". In: *International Conference on Automated Planning and Scheduling*. 2020, pp. 184–192.

[112] Edward Lam, Pierre Le Bodic, Daniel Damir Harabor, and Peter J Stuckey. "Branch-and-cut-and-price for multi-agent pathfinding." In: *International Joint Conference on Artificial Intelligence*. 2019, pp. 1289–1296.

[113] Ailsa H Land and Alison G Doig. "An automatic method for solving discrete programming problems". In: *50 Years of Integer Programming 1958-2008*. Springer, 2010, pp. 105–132.

[114] Jasmina Lazić, Saïd Hanafi, Nenad Mladenović, and Dragan Urošević. "Variable neighbourhood decomposition search for 0–1 mixed integer programs". In: *Computers & Operations Research* 37.6 (2010), pp. 1055–1067.

[115] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. "Towards a universal test suite for combinatorial auction algorithms". In: *ACM conference on Electronic Commerce*. 2000, pp. 66–76.

[116] Hui Li, Teng Long, Guangtong Xu, and Yangjie Wang. "Coupling-degree-based heuristic prioritized planning method for UAV swarm path generation". In: *Chinese Automation Congress*. 2019, pp. 3636–3641.

[117] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J Stuckey, and Sven Koenig. "Anytime multi-agent path finding via large neighborhood search". In: *International Joint Conference on Artificial Intelligence*. 2021, pp. 4127–4135.

[118] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. "Anytime multi-agent path finding via large neighborhood search: Extended abstract". In: *International Joint Conference on Autonomous Agents and Multiagent Systems*. 2021.

[119] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. "MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search". In: *AAAI Conference on Artificial Intelligence*. 2022, pp. 10256–10265.

[120] Jiaoyang Li, Ariel Felner, Eli Boyarski, Hang Ma, and Sven Koenig. "Improved heuristics for multi-agent path finding with conflict-based search." In: *International Joint Conference on Artificial Intelligence*. 2019, pp. 442–449.

[121] Jiaoyang Li, Graeme Gange, Daniel Harabor, Peter J Stuckey, Hang Ma, and Sven Koenig. "New techniques for pairwise symmetry breaking in multi-agent path finding". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. 2020, pp. 193–201.

[122] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. "Disjoint splitting for multi-agent path finding with conflict-based search". In: *International Conference on Automated Planning and Scheduling*. 2019, pp. 279–283.

[123] Jiaoyang Li, Eugene Lin, Hai L Vu, Sven Koenig, et al. "Intersection coordination with priority-based search for autonomous vehicles". In: *AAAI Conference on Artificial Intelligence*. Vol. 37. 10. 2023, pp. 11578–11585.

[124] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. "Eecbs: A bounded-suboptimal search for multi-agent path finding". In: *AAAI Conference on Artificial Intelligence*. Vol. 35. 14. 2021, pp. 12353–12362.

[125] Jiaoyang Li, Kexuan Sun, Hang Ma, Ariel Felner, TK Kumar, and Sven Koenig. "Moving agents in formation in congested environments". In: *Symposium on Combinatorial Search*. Vol. 11. 1. 2020, pp. 131–132.

[126] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W Durham, TK Satish Kumar, and Sven Koenig. "Lifelong multi-agent path finding in large-scale warehouses". In: *AAAI Conference on Artificial Intelligence*. Vol. 35. 13. 2021, pp. 11272–11281.

[127] Sirui Li, Zhongxia Yan, and Cathy Wu. "Learning to delegate for large-scale vehicle routing". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 26198–26211.

[128] Wenhao Li, Hongjun Chen, Bo Jin, Wenzhe Tan, Hongyuan Zha, and Xiangfeng Wang. "Multi-agent path finding with prioritized communication learning". In: *International Conference on Robotics and Automation*. IEEE. 2022, pp. 10695–10701.

[129]   Xiang Li, Tiejian Li, Jiahua Wei, Guangqian Wang, and William W-G Yeh. "Hydro unit commitment via mixed integer linear programming: A case study of the three gorges project, China". In: *IEEE Transactions on Power Systems* 29.3 (2013), pp. 1232–1241.

[130]   Zhuwen Li, Qifeng Chen, and Vladlen Koltun. "Combinatorial optimization with graph convolutional networks and guided tree search". In: *Advances in Neural Information Processing Systems* 31 (2018).

[131]   Defeng Liu, Matteo Fischetti, and Andrea Lodi. "Learning to search in local branching". In: *AAAI Conference on Artificial Intelligence*. 2022.

[132]   Hao Lu, Xingwen Zhang, and Shuang Yang. "A learning-based iterative method for solving vehicle routing problems". In: *International Conference on Learning Representations*. 2019.

[133]   Hao Lu, Xingwen Zhang, and Shuang Yang. "A learning-based iterative method for solving vehicle routing problems". In: *International Conference on Learning Representations*. 2020.

[134]   Paramet Luathep, Agachai Sumalee, William HK Lam, Zhi-Chun Li, and Hong K Lo. "Global optimization method for mixed transportation network design problem: a mixed-integer linear programming approach". In: *Transportation Research Part B: Methodological* 45.5 (2011), pp. 808–827.

[135]   Ryan Luna and Kostas E Bekris. "Push and swap: Fast cooperative path-finding with completeness guarantees". In: *International Joint Conference on Artificial Intelligence*. 2011, pp. 294–300.

[136]   Yuh-Chyun Luo, Monique Guignard, and Chun-Hung Chen. "A hybrid approach for integer programming combining genetic algorithms, linear programming and ordinal optimization". In: *Journal of Intelligent Manufacturing* 12 (2001), pp. 509–519.

[137]   Hang Ma, Daniel Harabor, Peter J. Stuckey, Jiaoyang Li, and Sven Koenig. "Searching with consistent prioritization for multi-agent path finding". In: *AAAI Conference on Artificial Intelligence*. 2019, pp. 7643–7650.

[138]   Hang Ma, Jiaoyang Li, TK Satish Kumar, and Sven Koenig. "Lifelong multi-agent path finding for online pickup and delivery tasks". In: *International Conference on Autonomous Agents and Multi-Agent Systems*. 2017, pp. 837–845.

[139]   Hang Ma, Craig Tovey, Guni Sharon, TK Satish Kumar, and Sven Koenig. "Multi-agent path finding with payload transfers and the package-exchange robot-routing problem". In: *AAAI Conference on Artificial Intelligence*. 2016.

[140]   Hang Ma, Jingxing Yang, Liron Cohen, TK Satish Kumar, and Sven Koenig. "Feasibility study: Moving non-homogeneous teams in congested video game environments". In: *Artificial Intelligence and Interactive Digital Entertainment Conference*. 2017, pp. 270–272.

[141]   Ziyuan Ma, Yudong Luo, and Hang Ma. "Distributed heuristic multi-agent path finding with communication". In: *2021 IEEE International Conference on Robotics and Automation*. IEEE. 2021, pp. 8699–8705.

[142] Stephen J Maher, Tobias Fischer, Tristan Gally, Gerald Gamrath, Ambros Gleixner, Robert Lion Gottwald, Gregor Hendel, Thorsten Koch, Marco Lübbecke, Matthias Miltenberger, et al. "The SCIP optimization suite 4.0". In: (2017).

[143] Sahil Manchanda, Akash Mittal, Anuj Dhawan, Sourav Medya, Sayan Ranu, and Ambuj Singh. "Learning heuristics over large graphs via deep reinforcement learning". In: *arXiv preprint arXiv:1903.03332* (2019).

[144] Alan S Manne. "On the job-shop scheduling problem". In: *Operations Research* 8.2 (1960), pp. 219–223.

[145] Renata Mansini, Wlodzimierz Ogryczak, and M Grazia Speranza. "Twenty years of linear programming based portfolio optimization". In: *European Journal of Operational Research* 234.2 (2014), pp. 518–535.

[146] Leilei Meng, Chaoyong Zhang, Yaping Ren, Biao Zhang, and Chang Lv. "Mixed-integer linear programming and constraint programming formulations for solving distributed flexible job shop scheduling problem". In: *Computers & Industrial Engineering* 142 (2020), p. 106347.

[147] Maxime Mulamba, Jayanta Mandi, Michelangelo Diligenti, Michele Lombardi, Victor Bucarey Lopez, and Tias Guns. "Contrastive losses and solution caching for predict-and-optimize". In: *International Joint Conference on Artificial Intelligence.* 2021, p. 2833.

[148] Vinod Nair, Sergey Bartunov, Felix Gimeno, et al. "Solving mixed integer programs using neural networks". In: *arXiv preprint arXiv:2012.13349* (2020).

[149] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takác. "Reinforcement learning for solving the vehicle routing problem". In: *Advances in Neural Information Processing Systems* 31 (2018).

[150] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. "Priority inheritance with backtracking for iterative multi-agent path finding". In: *Artificial Intelligence* 310 (2022), p. 103752.

[151] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. "Representation learning with contrastive predictive coding". In: *arXiv preprint arXiv:1807.03748* (2018).

[152] James Ostrowski, Miguel F Anjos, and Anthony Vannelli. "Tight mixed integer linear programming formulations for the unit commitment problem". In: *IEEE Transactions on Power Systems* 27.1 (2011), pp. 39–46.

[153] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity.* Courier Corporation, 1998.

[154] Max B Paulus, Giulia Zarpellon, Andreas Krause, Laurent Charlin, and Chris Maddison. "Learning to cut by looking ahead: Cutting plane selection via imitation learning". In: *International Conference on Machine Learning.* PMLR. 2022, pp. 17584–17600.

[155] Judea Pearl and Jin H. Kim. "Studies in semi-admissible heuristics". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 4 (1982), pp. 392–399.

[156] Thomy Phan, Taoan Huang, Bistra Dilkina, and Sven Koenig. "Adaptive anytime multi-agent path finding using bandit-based large neighborhood search". In: *AAAI Conference on Artificial Intelligence*. Vol. 38. 16. 2024, pp. 17514–17522.

[157] Victor Pillac, Pascal Van Hentenryck, and Caroline Even. "A conflict-based path-generation heuristic for evacuation planning". In: *Transportation Research Part B: Methodological* 83 (2016), pp. 136–150.

[158] Ira Pohl. "Heuristic search viewed as path finding in a graph". In: *Artificial intelligence* 1.3-4 (1970), pp. 193–204.

[159] Antoine Prouvost, Justin Dumouchelle, Lara Scavuzzo, Maxime Gasse, Didier Chételat, and Andrea Lodi. "Ecole: A gym-like library for machine learning in combinatorial optimization solvers". In: *Learning Meets Combinatorial Algorithms at NeurIPS2020*. 2020. URL: https://openreview.net/forum?id=IVc9hqgibyB.

[160] Qi Qian, Lei Shang, Baigui Sun, Juhua Hu, Hao Li, and Rong Jin. "Softtriple loss: Deep metric learning without triplet sampling". In: *IEEE/CVF International Conference on Computer Vision*. 2019, pp. 6450–6458.

[161] Arthur Queffelec, Ocan Sankur, and François Schwarzentruber. "Conflict-based search for connected multi-agent path finding". In: *arXiv preprint arXiv:2006.03280* (2020).

[162] C Quoc and Viet Le. "Learning to rank with nonsmooth cost functions". In: *Advances in Neural Information Processing Systems*. 2007, pp. 193–200.

[163] Jingyao Ren, Vikraman Sathiyanarayanan, Eric Ewing, Baskin Senbaslar, and Nora Ayanian. "MAPFAST: A deep algorithm selector for multi agent path finding using shortest path embeddings". In: *International Conference on Autonomous Agents and MultiAgent Systems*. 2021.

[164] Nils Rethmeier and Isabelle Augenstein. "A primer on contrastive pretraining in language processing: Methods, lessons learned, and perspectives". In: *ACM Computing Surveys* 55.10 (2023), pp. 1–17.

[165] Julia Rieck, Juergen Zimmermann, and Thorsten Gather. "Mixed-integer linear programming for resource leveling problems". In: *European Journal of Operational Research* 221.1 (2012), pp. 27–37.

[166] Stefan Ropke and David Pisinger. "An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows". In: *Transportation science* 40.4 (2006), pp. 455–472.

[167] Stéphane Ross and Drew Bagnell. "Efficient reductions for imitation learning". In: *International Conference on Artificial Intelligence and Statistics*. 2010, pp. 661–668.

[168] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. "A reduction of imitation learning and structured prediction to no-regret online learning". In: *International Conference on Artificial Intelligence and Statistics*. 2011, pp. 627–635.

[169]   Edward Rothberg. "An evolutionary algorithm for polishing mixed integer programming solutions". In: *INFORMS Journal on Computing* 19.4 (2007), pp. 534–541.

[170]   Qandeel Sajid, Ryan Luna, and Kostas Bekris. "Multi-agent pathfinding with simultaneous execution of single-agent primitives". In: *Symposium on Combinatorial Search*. Vol. 3. 1. 2012, pp. 88–96.

[171]   Arun Kumar Sangaiah, Erfan Babaee Tirkolaee, Alireza Goli, and Saeed Dehnavi-Arani. "Robust optimization and mixed-integer linear programming model for LNG supply chain planning problem". In: *Soft Computing* 24 (2020), pp. 7885–7905.

[172]   Guillaume Sartoretti, Justin Kerr, Yunfei Shi, Glenn Wagner, TK Satish Kumar, Sven Koenig, and Howie Choset. "PRIMAL: Pathfinding via reinforcement and imitation multi-agent learning". In: *IEEE Robotics and Automation Letters* 4.3 (2019), pp. 2378–2385.

[173]   Lara Scavuzzo, Feng Chen, Didier Chételat, Maxime Gasse, Andrea Lodi, Neil Yorke-Smith, and Karen Aardal. "Learning to branch with tree MDPs". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 18514–18526.

[174]   Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. "Conflict-based search for optimal multi-agent pathfinding". In: *Artificial Intelligence* 219 (2015), pp. 40–66.

[175]   David Silver. "Cooperative pathfinding". In: *Artificial Intelligence and Interactive Digital Entertainment Conference*. 2005, pp. 117–122.

[176]   Stephen L Smith and Frank Imeson. "GLNS: An effective large neighborhood search heuristic for the generalized traveling salesman problem". In: *Computers & Operations Research* 87 (2017), pp. 1–19.

[177]   Jialin Song, Ravi Lanka, Yisong Yue, and Bistra Dilkina. "A general large neighborhood search framework for solving integer linear programs". In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020.

[178]   Jialin Song, Ravi Lanka, Albert Zhao, Aadyot Bhatnagar, Yisong Yue, and Masahiro Ono. "Learning to search via retrospective imitation". In: *arXiv preprint arXiv:1804.00846* (2018).

[179]   Nicolas Sonnerat, Pengming Wang, Ira Ktena, Sergey Bartunov, and Vinod Nair. "Learning a large neighborhood search algorithm for mixed integer programs". In: *arXiv preprint arXiv:2107.10201* (2021).

[180]   Trevor Scott Standley. "Finding optimal solutions to cooperative pathfinding problems." In: *AAAI Conference on Artificial Intelligence*. 2010, pp. 28–29.

[181]   Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Satish Kumar, Eli Boyarski, and Roman Bartak. "Multi-agent pathfinding: Definitions, variants, and benchmarks". In: *Symposium on Combinatorial Search*. 2019, pp. 151–158.

[182] Nathan R. Sturtevant. "Benchmarks for grid-based pathfinding". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.2 (2012), pp. 144–148.

[183] Pavel Surynek. "Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories". In: *Symposium on Combinatorial Search*. Vol. 10. 1. 2019, pp. 202–203.

[184] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. "Reinforcement learning for integer programming: Learning to cut". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 9367–9376.

[185] J Teghem, M Pirlot, and C Antoniadis. "Embedding of linear programming in a simulated annealing algorithm for solving a mixed integer production planning problem". In: *Journal of Computational and Applied Mathematics* 64.1-2 (1995), pp. 91–102.

[186] Jordan Tyler Thayer and Wheeler Ruml. "Bounded suboptimal search: A direct approach using inadmissible estimates". In: *International Joint Conference on Artificial Intelligence*. Vol. 2011. 2011, pp. 674–679.

[187] Yuandong Tian. "Understanding Deep Contrastive Learning via Coordinate-wise Optimization". In: *Advances in Neural Information Processing Systems*. 2022.

[188] Zekun Tong, Yuxuan Liang, Henghui Ding, Yongxing Dai, Xinke Li, and Changhu Wang. "Directed graph contrastive learning". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 19580–19593.

[189] Paolo Toth and Daniele Vigo. *The vehicle routing problem*. SIAM, 2002.

[190] Glenn Wagner and Howie Choset. "M*: A complete multirobot path planning algorithm with performance bounds". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2011, pp. 3260–3267.

[191] Jiangxing Wang, Jiaoyang Li, Hang Ma, Sven Koenig, and S Kumar. "A new constraint satisfaction perspective on multi-agent path finding: Preliminary results". In: *International Conference on Autonomous Agents and Multiagent Systems*. 2019, pp. 2253–2255.

[192] Shuwei Wang, Vadim Bulitko, Taoan Huang, Sven Koenig, and Roni Stern. "Synthesizing priority planning formulae for multi-agent pathfinding". In: *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 19. 1. 2023, pp. 360–369.

[193] Yutong Wang, Bairan Xiang, Shinan Huang, and Guillaume Sartoretti. "SCRIMP: Scalable communication for reinforcement-and imitation-learning-based multi-agent pathfinding". In: *International Conference on Autonomous Agents and Multiagent Systems*. 2023, pp. 2598–2600.

[194] Laurence A. Wolsey and George L. Nemhauser. *Integer and combinatorial optimization*. Vol. 55. John Wiley & Sons, 1999.

[195] Chao-Yuan Wu, R Manmatha, Alexander J Smola, and Philipp Krahenbuhl. "Sampling matters in deep embedding learning". In: *IEEE International Conference on Computer Vision*. 2017, pp. 2840–2848.

[196] Wenying Wu, Subhrajit Bhattacharya, and Amanda Prorok. "Multi-robot path deconfliction through prioritization by path prospects". In: *IEEE International Conference on Robotics and Automation*. 2020, pp. 9809–9815.

[197] Wenying Wu, Subhrajit Bhattacharya, and Amanda Prorok. "Multi-robot path deconfliction through prioritization by path prospects". In: *IEEE international conference on robotics and automation*. IEEE. 2020, pp. 9809–9815.

[198] Yaoxin Wu, Wen Song, Zhiguang Cao, and Jie Zhang. "Learning large neighborhood search policy for integer programming". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 30075–30087.

[199] Peter R Wurman, Raffaello D'Andrea, and Mick Mountz. "Coordinating hundreds of cooperative, autonomous vehicles in warehouses". In: *AI Magazine* 29.1 (2008), pp. 9–9.

[200] Liang Xin, Wen Song, Zhiguang Cao, and Jie Zhang. "NeuroLKH: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 7472–7483.

[201] Zhongxia Yan and Cathy Wu. "Neural neighborhood search for multi-agent path finding". In: *International Conference on Learning Representations*. 2024.

[202] Yu Yang, Natashia Boland, Bistra Dilkina, and Martin Savelsbergh. "Learning generalized strong branching for set covering, set packing, and 0–1 knapsack problems". In: *European Journal of Operational Research* 301.3 (2022), pp. 828–840.

[203] Taehyun Yoon, Jinwon Choi, Hyokun Yun, and Sungbin Lim. "Threshold-aware Learning to Generate Feasible Solutions for Mixed Integer Programs". In: *arXiv preprint arXiv:2308.00327* (2023).

[204] Fengqi You and Ignacio E Grossmann. "Mixed-integer nonlinear programming models and algorithms for large-scale supply chain design with stochastic inventory management". In: *Industrial & Engineering Chemistry Research* 47.20 (2008), pp. 7802–7817.

[205] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. "Graph contrastive learning with augmentations". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 5812–5823.

[206] Chenning Yu, Qingbiao Li, Sicun Gao, and Amanda Prorok. "Accelerating multi-agent planning using graph transformers with bounded suboptimality". In: *IEEE International Conference on Robotics and Automation*. 2023, pp. 3432–3439.

[207] Jingjin Yu. "Intractability of optimal multirobot path planning on planar graphs". In: *IEEE Robotics and Automation Letters* 1.1 (2015), pp. 33–40.

[208] Jingjin Yu and Steven M LaValle. "Structure and intractability of optimal multi-robot path planning on graphs". In: *AAAI Conference on Artificial Intelligence*. 2013, pp. 1443–1449.

[209] Jingjin Yu and Steven M. LaValle. "Planning optimal paths for multiple robots on graphs". In: *IEEE International Conference on Robotics and Automation*. 2013, pp. 3612–3617.

[210] Jingjin Yu and Daniela Rus. "Pebble motion on graphs with rotations: Efficient feasibility tests and planning algorithms". In: *Algorithmic Foundations of Robotics XI: Selected Contributions of the Eleventh International Workshop on the Algorithmic Foundations of Robotics*. Springer. 2015, pp. 729–746.

[211] Giulia Zarpellon, Jason Jo, Andrea Lodi, and Yoshua Bengio. "Parameterizing branch-and-bound search trees to learn branching policies". In: *AAAI Conference on Artificial Intelligence*. 2021.

[212] Jiayi Zhang, Chang Liu, Xijun Li, Hui-Ling Zhen, Mingxuan Yuan, Yawen Li, and Junchi Yan. "A survey for solving mixed integer programming via machine learning". In: *Neurocomputing* 519 (2023), pp. 205–217.

[213] Shuyang Zhang, Jiaoyang Li, Taoan Huang, Sven Koenig, and Bistra Dilkina. "Learning a priority ordering for prioritized planning in multi-agent path finding". In: *Symposium on Combinatorial Search*. 2022.

[214] Jiongzhi Zheng, Kun He, Jianrong Zhou, Yan Jin, and Chu-Min Li. "Combining reinforcement learning with Lin-Kernighan-Helsgaun algorithm for the traveling salesman problem". In: *AAAI Conference on Artificial Intelligence*. 2021.

[215] Ivan Žulj, Sergej Kramer, and Michael Schneider. "A hybrid of adaptive large neighborhood search and tabu search for the order-batching problem". In: *European Journal of Operational Research* 264.2 (2018), pp. 653–664.

Appendix

A Supplementary Materials to Chapter 3

A.1 Additional Details of MILP Instance Generation

We present the MILP formulations for the minimum vertex cover (MVC), maximum independent set (MIS), set covering (SC) and combinatorial auction (CA) problems. The descriptions and formulations for the item placement and workload appointment problems can be found at the ML4CO competition [58] website[*].

In an MVC instance, we are given an undirected graph $G = (V, E)$. The goal is to select the smallest subset of nodes such that at least one end point of every edge in the graph is selected:

$$\min \sum_{v \in V} x_v$$

$$\text{s.t.} \quad x_u + x_v \geq 1, \ \forall (u, v) \in E,$$

$$x_v \in \{0, 1\}, \ \forall v \in V.$$

---

[*]ML4CO Competition Website: https://github.com/ds4dm/ml4co-competition/blob/main/DATA.md

In an MIS instance, we are given an undirected graph $G = (V, E)$. The goal is to select the largest subset of nodes such that no two nodes in the subsets are connected by an edge in G:

$$\min - \sum_{v \in V} x_v$$

$$\text{s.t.} \quad x_u + x_v \leq 1, \; \forall (u, v) \in E,$$

$$x_v \in \{0, 1\}, \; \forall v \in V.$$

In an SC instance, we are given m elements and a collection S of sets whose union is the set of all elements. The goal is to select a minimum number of sets from S such that the union of the selected set is the set of all elements:

$$\min \sum_{s \in S} x_s$$

$$\text{s.t.} \quad \sum_{s \in S : i \in s} x_s \geq 1, \; \forall i \in [m],$$

$$x_s \in \{0, 1\}, \; \forall s \in S.$$

In a CA instance, we are given $\tilde{n}$ bids $\{(B_i, p_i) : i \in [\tilde{n}]\}$ for $\tilde{m}$ items, where B_i is a subset of items and p_i is its associated bidding price. The objective is to allocate items to bids such that the total revenue is maximized:

$$\min - \sum_{i \in [\tilde{n}]} p_i x_i$$

$$\text{s.t.} \quad \sum_{i : j \in B_i} x_i \leq 1, \; \forall j \in [\tilde{m}],$$

$$x_i \in \{0, 1\}, \; \forall i \in [\tilde{n}].$$

A.2 Supplementary Materials to Section 3.6

A.2.1 Neural Network Architecture for CL-LNS

We give full details of the GAT architecture described in subsection 3.6.1.2. The policy takes as input the state s^t and outputs a score vector $\pi_{\boldsymbol{\theta}}(s^t) \in [0,1]^n$, one score per variable. We use 2-layer MLPs with 64 hidden units per layer and ReLU as the activation function to map each node feature and edge feature to $\mathbb{R}^d$ where $d = 64$.

Let $\mathbf{v}_j, \mathbf{c}_i, \mathbf{e}_{i,j} \in \mathbb{R}^d$ be the embeddings of the j-th variable, i-th constraint and the edge connecting them output by the embedding layers. We perform two rounds of message passing through the GAT. In the first round, each constraint node $\mathbf{c}_i$ attends to its neighbors $\mathcal{N}_i$ using an attention structure with $H = 8$ attention heads:

$$\mathbf{c}_i' = \frac{1}{H} \sum_{h=1}^{H} \left(\alpha_{ii,1}^{(h)} \boldsymbol{\theta}_{c,1}^{(h)} \mathbf{c}_i + \sum_{j \in \mathcal{N}_i} \alpha_{ij,1}^{(h)} \boldsymbol{\theta}_{v,1}^{(h)} \mathbf{v}_j \right)$$

where $\boldsymbol{\theta}_{c,1}^{(h)} \in \mathbb{R}^{d \times d}$ and $\boldsymbol{\theta}_{v,1}^{(h)} \in \mathbb{R}^{d \times d}$ are learnable weights. The updated constraint embeddings $\mathbf{c}_i'$ are averaged across H attention heads using attention weights [23]

$$\alpha_{ij,1}^{(h)} = \frac{\exp(\boldsymbol{w}_1^{\mathsf{T}} \rho([\boldsymbol{\theta}_{c,1}^{(h)} \mathbf{c}_i, \boldsymbol{\theta}_{v,1}^{(h)} \mathbf{v}_j, \boldsymbol{\theta}_{e,1}^{(h)} \mathbf{e}_{i,j}]))}{\sum_{k \in \mathcal{N}_i} \exp(\boldsymbol{w}_1^{\mathsf{T}} \rho([\boldsymbol{\theta}_{c,1}^{(h)} \mathbf{c}_i, \boldsymbol{\theta}_{v,1}^{(h)} \mathbf{v}_k, \boldsymbol{\theta}_{e,1}^{(h)} \mathbf{e}_{i,k}]))}$$

where the attention coefficients $\boldsymbol{w}_1 \in \mathbb{R}^{3d}$ and $\boldsymbol{\theta}_{e,1}^{(h)} \in \mathbb{R}^{d \times d}$ are both learnable weights and $\rho(\cdot)$ refers to the LeakyReLU activation function with negative slope 0.2. In the second round, similarly, each variable node attends to its neighbors to get updated variable node embeddings

$$\mathbf{v}_j' = \frac{1}{H} \sum_{h=1}^{H} \left(\alpha_{jj,2}^{(h)} \boldsymbol{\theta}_{v,2}^{(h)} \mathbf{v}_j + \sum_{i \in \mathcal{N}_j} \alpha_{ji,2}^{(h)} \boldsymbol{\theta}_{c,2}^{(h)} \mathbf{c}_i' \right)$$

with attention weights

$$\alpha_{ji,2}^{(h)} = \frac{\exp(\boldsymbol{w}_2^\mathsf{T} \rho([\boldsymbol{\theta}_{c,2}^{(h)}\mathbf{c}_i', \boldsymbol{\theta}_{v,2}^{(h)}\mathbf{v}_j, \boldsymbol{\theta}_{e,2}^{(h)}\mathbf{e}_{i,j}]))}{\sum_{k\in\mathcal{N}_j} \exp(\boldsymbol{w}_2^\mathsf{T} \rho([\boldsymbol{\theta}_{c,2}^{(h)}\mathbf{c}_i', \boldsymbol{\theta}_{v,2}^{(h)}\mathbf{v}_j, \boldsymbol{\theta}_{e,2}^{(h)}\mathbf{e}_{i,k}]))}$$

where $\boldsymbol{w}_2 \in \mathbb{R}^{3d}$ and $\boldsymbol{\theta}_{c,2}^{(h)}, \boldsymbol{\theta}_{v,2}^{(h)}, \boldsymbol{\theta}_{e,2}^{(h)} \in \mathbb{R}^{d\times d}$ are learnable weights. After the two rounds of message passing, the final representations of variables $\mathbf{v}'$ are passed through a 2-layer MLP with 64 hidden units per layer to obtain a scalar value for each variable. Finally, we apply the sigmoid function to get a score between 0 and 1.

Features We use features proposed in [59] for node features and edge features in the bipartite graph and also include a fixed-size window of most recent incumbent values as variable node features with the window size set to 3 in experiments. In addition, we include features proposed in [102] computed at the root node of BnB to make it a richer set of variable node features. The full list of features can be found in Table 2 in Appendix of [59] and Table 1 in [102]. In our implementation, we compute them using the APIs provided by the Ecole library [159][†].

A.2.2 Hyperparameter Tuning

For RL-LNS, we use all the hyperparameters provided in their code [198] in our experiments. For the other LNS methods, all hyperparameters used in experiments are fine-tuned on the validation set, and the hyperparameter tunings are described below.

For β, which upper bounds the neighborhood size, we tried values from $\{0.25, 0.5, 0.6, 0.7\}$. $\beta = 0.25$ is the worst for all approaches, resulting in the highest gap. For LB-RELAX, IL-LNS and CL-LNS, all values perform similarly (because they select effective neighborhoods early in the search and their neighborhood

---

[†]More details and the source code can be found at https://doc.ecole.ai/py/en/stable/reference/observations.html.

sizes either do not reach the upper bound or they already converge to good solutions before reaching it).

For RANDOM and GRAPH, $\beta = 0.5$ is the best for them. So, we set $\beta = 0.5$ consistently for all approaches.

For initial neighborhood sizes k^0, we observe that the best values are sensitive for approaches that need longer runtime to select variables, such as LB-RELAX, IL-LNS and CL-LNS, thus they need the right k^0 from the beginning and we fine-tune it for them. For RANDOM and GRAPH, their runtime for selecting variables is short, and with the adaptive neighborhood size mechanism, they could very quickly find the right neighborhood size and are insensitive to k^0. They converge to the same primal gaps ($< 1\%$ relative differences) with similar primal integrals ($< 2\%$ relative differences) using different k^0. Despite the differences being small, we still use the best k^0 for them.

For γ that controls the rate at which k^t increases, we tried values from $\{1, 1.01, 1.02, 1.05\}$. Overall, γ does not greatly impact the performance if $\gamma > 1$; however, $\gamma = 1$ is far worse than the others.

For the runtime limit for each repair operation, we tried different limits of 0.5, 1, 2 and 5 minutes. None of the approaches are sensitive to it since most repairs are finished within 20 seconds. Except for IL-LNS on the SC instances, it selects neighborhoods that require a longer time to repair and a 2-minute runtime limit is necessary. Therefore, we use 2 minutes consistently.

For BnB, the aggressive mode is fine-tuned for each problem on the validation set. With the aggressive mode turned on, BnB (SCIP) does not always deliver better anytime performance compared to when it is turned off. Based on the validation results, the aggressive mode is turned on for MVC and SC instances and turned off for CAT and MIS instances.

For IL-LNS, it uses the same training dataset as CL-LNS but uses only the positive samples. We fine-tune its hyperparameters for each problem on the validation set, resulting in a different k^0 on the SC instance from CL-LNS. In [179], they use sampling methods to select variables when using the learned policy. For the temperature parameter η in the sampling method, we tried values from $\{1/2, 2/3, 1\}$ and $\eta = 0.5$ performs the best overall. However, in our experiment, we observe that our greedy method described in

Table A.1: Hyperparameters with their notations and values used.

Hyperparameter	Notation	Value
Suboptimality threshold to determine positive samples	α_{p}	0.5
Upper bound on the number of positive samples	u_{p}	10
Suboptimality threshold to determine negative samples	α_{n}	0.05
Ratio between the numbers of positive and negative samples	κ	9
Feature embedding dimension	d	64
Window size of the most recent incumbent values in variable features		3
Number of attention heads in the GAT	H	8
Temperature parameter in the contrastive loss	τ	0.07
Rate at which k^t increases	γ	1.02
Upper bound on k^t as a fraction of number of variables	β	0.5
Temperature parameter for sampling variables in IL-LNS	η	0.5
Initial neighborhood size	k^0	Fine-tuned for each case
Runtime for finding initial solution		10 seconds
Runtime limit for each reoptimization		2 minutes
Learning rate for CL-LNS and IL-LNS		10^{-3}
Batch size for CL-LNS and IL-LNS		32
Number of training epochs for CL-LNS and IL-LNS		30

subsection 3.6.1.4 works better for IL-LNS on SC and MIS instances. Thus, CL-LNS is compared against the corresponding results on SC and MIS instances.

For LB-RELAX, three variants are presented in [86]. For simplicity, we present only the best of the three variants for each problem in the paper.

In Table A.1, we summarize all the hyperparameters with their notations and values used in our experiments.

A.2.3   Additional Experimental Results

In this subsection, we add two more baselines and evaluate all approaches on one more metric. We show that CL-LNS outperforms all approaches in terms of all metrics.

We establish two additional baselines:

- **LB** LNS which selects the neighborhood with the LB heuristics. We set the time limit to 10 minutes for solving the LB ILP in each iteration;

- **GRAPH** LNS which selects the neighborhood based on the bipartite graph representation of the ILP similar to GINS [142]. A bipartite graph representation consists of nodes representing the variables and constraints on two sides, respectively, with an edge connecting a variable and a constraint if a variable has a non-zero coefficient in the constraint. It runs a breadth-first search starting from a random variable node in the bipartite graph and selects the first k^t variable nodes expanded.

Figure A.1 shows the full results on the primal gap as a function of runtime. Figure A.2 shows the full results on the survival rate as a function of runtime. Figure A.3 shows the full results on the primal bound as a function of runtime. Tables A.2 and A.3 present the average primal bound, primal gap and primal integral at 30 and 60 minutes runtime cutoff, respectively, on the small instances. Tables A.4 and A.5 present the average primal bound, primal gap and primal integral at 30 and 60 minutes runtime cutoff, respectively, on the large instances.

Next, we evaluate the performance with one additional metric: The *gap to virtual best* at time z for an approach is the normalized difference between its best primal bound found up to time z and the best primal bound found up to time z by any approach in the portfolio.

Figure A.4 shows the full results on the best performing rate as a function of runtime. Figure A.5 shows the full results on the gap to virtual best as a function of runtime.

## A.3 Supplementary Materials to Section 3.7

A.3.1 Neural Network Architecture for ConPaS

We follow previous work [59, 70] to use a bipartite graph representation to encode a MILP M. For the node (variable and constraint)and edge features of the bipartite graph, we use the same features as [70].
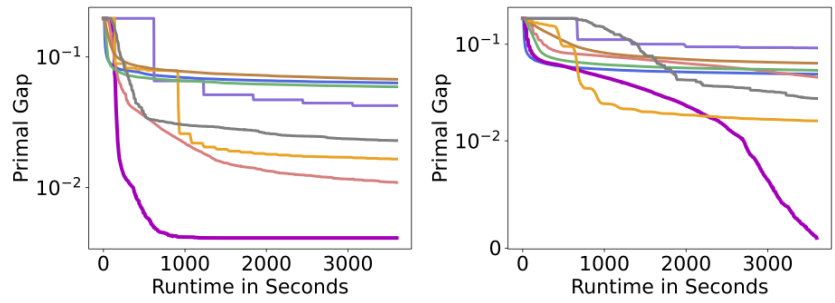
We use the same GCN architecture as previous work [70]. The GCN takes as input the bipartite graph representation of a MILP M with its features and outputs $p_\theta(x|M)$, a $[0, 1]$-score vector for the binary variables. For node features, we use 2-layer multi-layer perceptrons (MLP) with 64 hidden units per layer
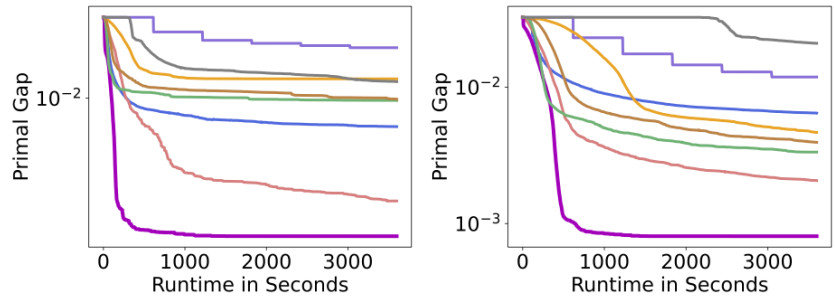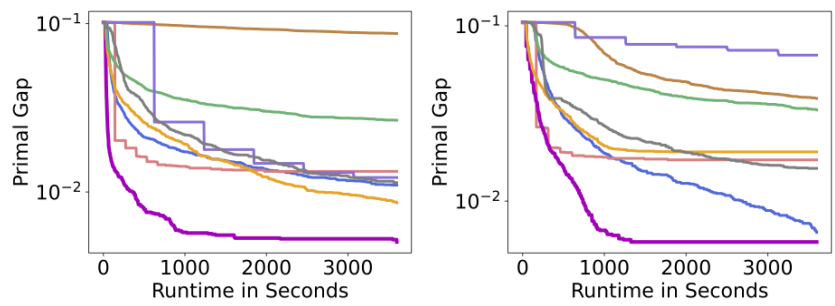
(a) MVC-S (left) and MVC-L (right).
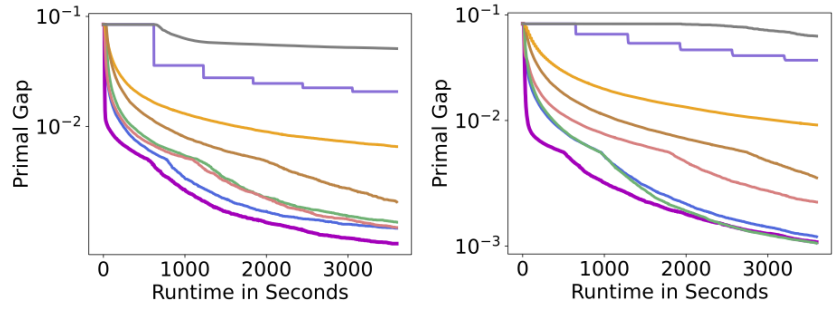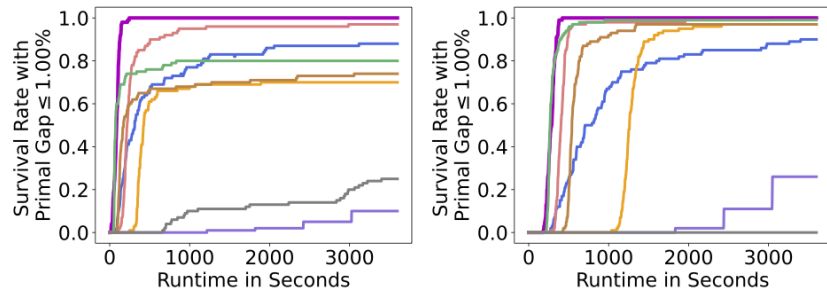
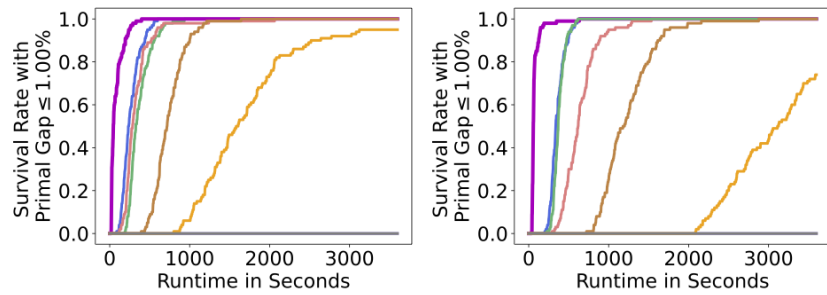(b) MIS-S (left) and MIS-L (right).

(c) CA-S (left) and CA-L (right).
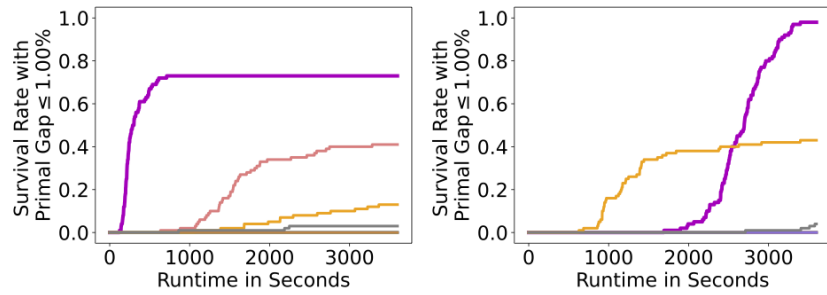
(d) SC-S (left) and SC-L (right).

Figure A.1: The primal gap (the lower the better) as a function of time, averaged over 100 instances. For ML approaches, the policies are trained on only small training instances but tested on both small and large test instances.

(a) MVC-S (left) and MVC-L (right).

(b) MIS-S (left) and MIS-L (right).

(c) CA-S (left) and CA-L (right).

(d) SC-S (left) and SC-L (right).

Figure A.2: The survival rate (the higher the better) over 100 instances as a function of time to meet primal gap threshold 1.00%. For ML approaches, the policies are trained on only small training instances but tested on both small and large test instances.
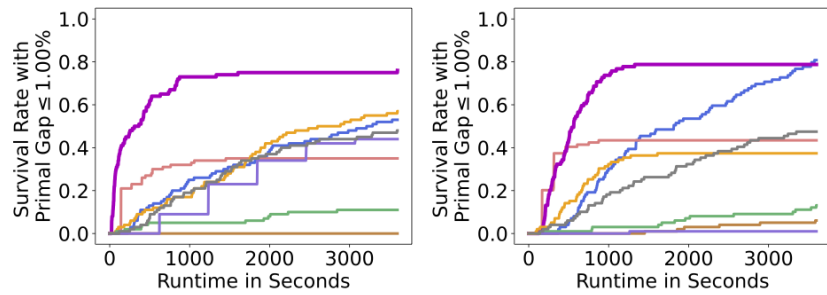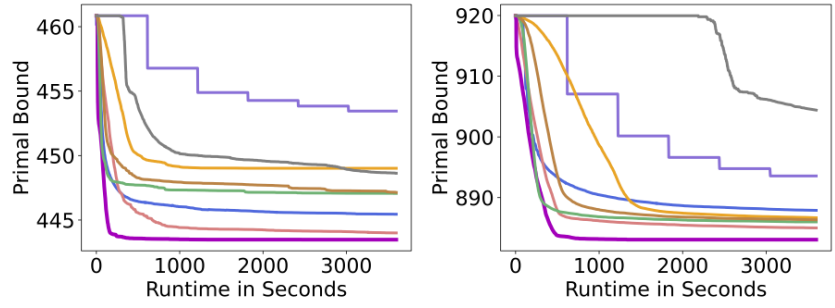
(a) MVC-S (left) and MVC-L (right).

(b) MIS-S (left) and MIS-L (right).

(c) CA-S (left) and CA-L (right).

(d) SC-S (left) and SC-L (right).

Figure A.3: The primal bound (the lower the better) as a function of time, averaged over 100 instances. For ML approaches, the policies are trained on only small training instances but tested on both small and large test instances.

(a) MVC-S (left) and MVC-L (right).

(b) MIS-S (left) and MIS-L (right).

(c) CA-S (left) and CA-L (right).

(d) SC-S (left) and SC-L (right).

Figure A.4: The best performing rate (the higher the better) as a function of runtime over 100 test instances. For ML approaches, the policies are trained on only small training instances but tested on both small and large test instances.

(a) MVC-S (left) and MVC-L (right).

(b) MIS-S (left) and MIS-L (right).
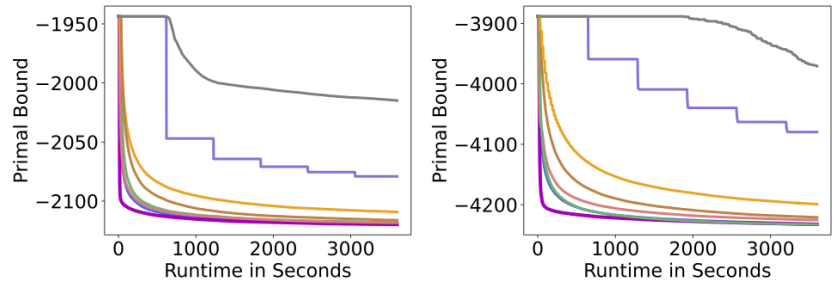

(c) CA-S (left) and CA-L (right).

(d) SC-S (left) and SC-L (right).
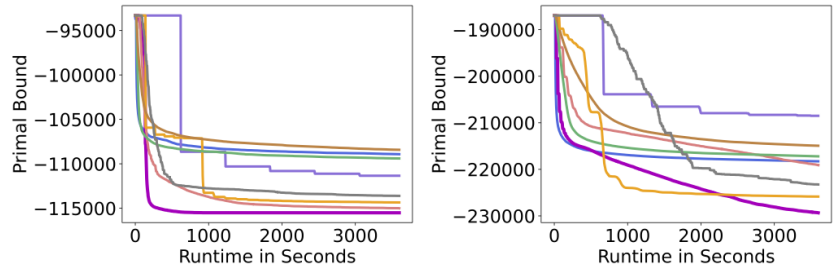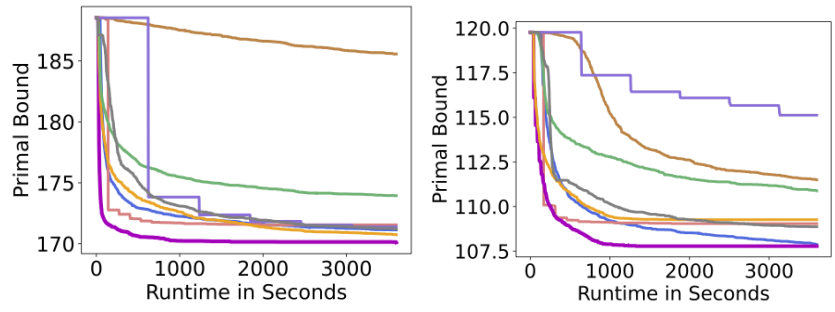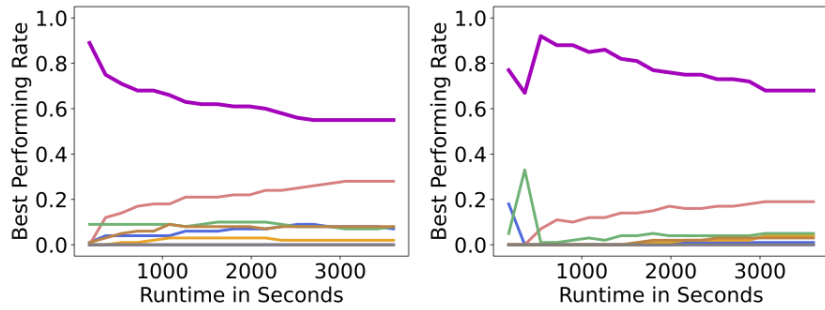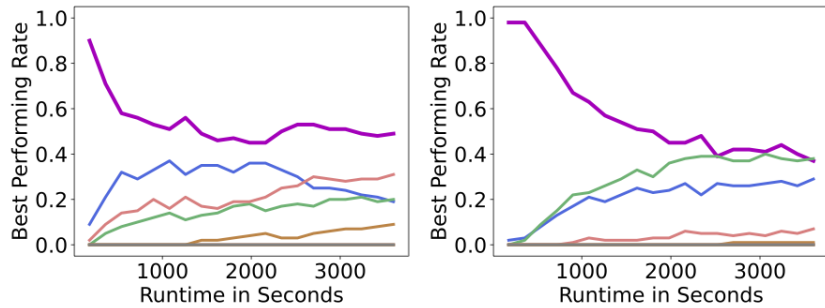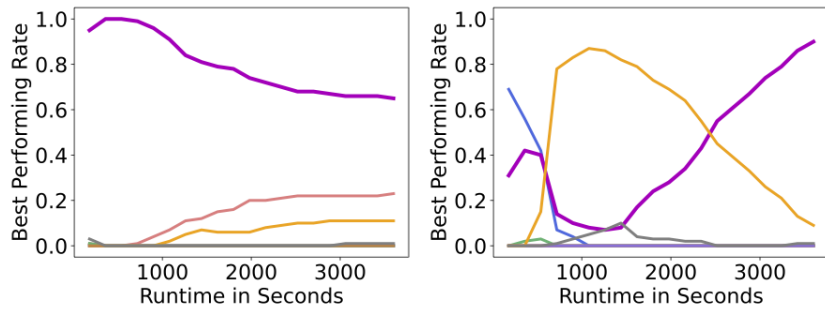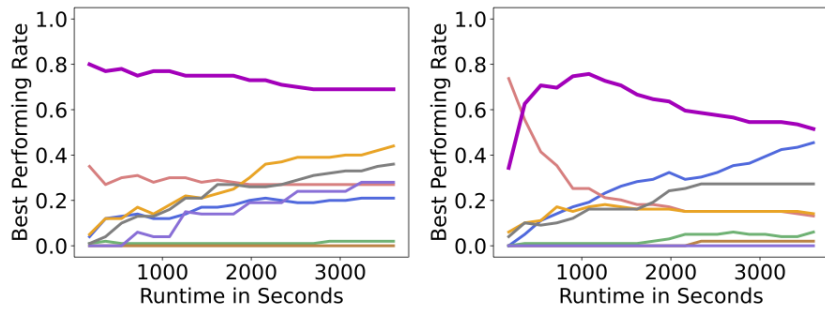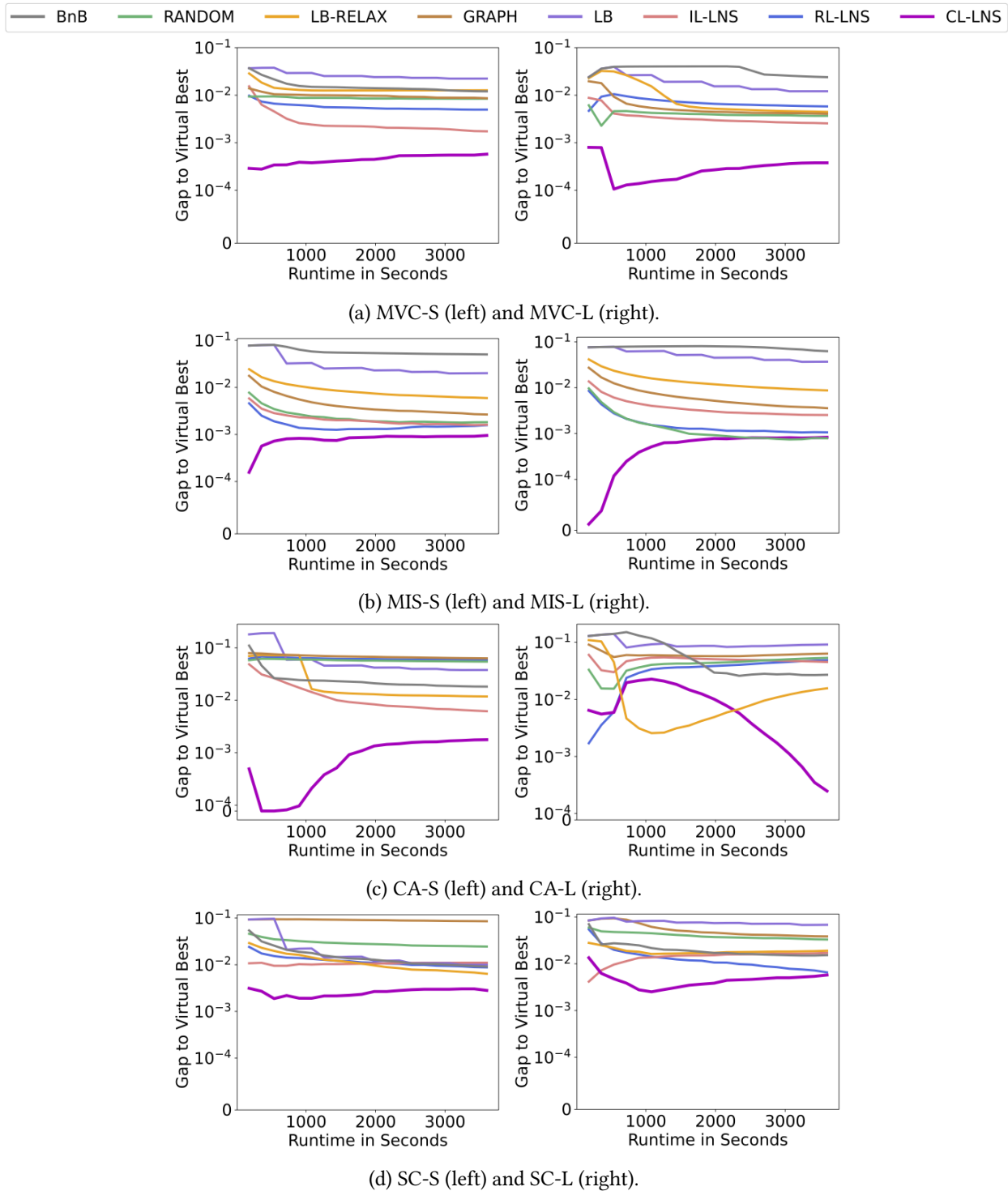
Figure A.5: The gap to virtual best (the lower the better) as a function of runtime, averaged over 100 test instances. For ML approaches, the policies are trained on only small training instances but tested on both small and large test instances.

	PB	PG (%)	PI	PB	PG (%)	PI
	MVC			MIS		
BnB	449.67±9.69	1.55±0.44	40.2±6.6	-2,004.24±26.21	5.60±1.00	127.1±12.4
LB	454.89±11.55	2.66±1.16	58.2±14.1	-2,064.30±16.40	2.77±0.51	89.9±7.3
RANDOM	447.16±11.22	0.98±1.26	20.6±22.5	-2,115.23±11.82	0.37±0.16	16.9±2.7
GRAPH	447.75±11.39	1.11±1.30	24.2±22.1	-2,111.84±12.06	0.53±0.16	24.4±2.7
LB-RELAX	449.02±11.53	1.38±1.51	32.1±24.2	-2,102.85±11.97	0.95±0.19	33.0±3.6
IL-LNS	444.27±9.61	0.35±0.25	13.5±6.9	-2,115.30±12.04	0.36±0.18	14.4±3.2
RL-LNS	445.71±9.98	0.67±0.35	18.2±5.7	-2,116.64±11.53	0.30±0.15	12.7±2.9
CL-LNS	**443.48±9.56**	**0.17±0.09**	**5.5±3.6**	**-2,117.58±11.86**	**0.26±0.17**	**9.3±3.0**
	CA			SC		
BnB	-113,068±1,595	2.75±0.62	93.5±18.6	172.09±12.65	1.63±1.20	62.9±22.5
LB	-110,303±2,001	5.13±1.08	191.6±16.9	172.37±12.71	1.79±1.11	89.4±22.3
RANDOM	-109,040±1,685	6.21±1.05	126.8±17.6	174.70±12.75	3.10±1.38	73.4±24.6
GRAPH	-107,802±1,892	7.28±1.07	152.2±18.9	186.79±14.13	9.33±2.28	175.7±38.8
LB-RELAX	-114,103±1,521	1.86±0.57	109.5±9.4	171.60±12.43	1.36±1.02	44.6±19.3
IL-LNS	-114,621±1638	1.41±0.58	68.1±13.9	171.59±12.45	1.35±1.00	39.3±17.4
RL-LNS	-108,562±1,854	6.63±1.05	132.9±18.2	171.70±12.30	1.42±0.88	55.7±15.6
CL-LNS	**-115,513±1,621**	**0.65±0.32**	**39.1±11.6**	**170.16±12.13**	**0.53±0.63**	**16.7±12.3**

Table A.2: Test results on small instances: Primal bound (PB), primal gap (PG) (in percent), primal integral (PI) at 30 minutes time cutoff, averaged over 100 instances and their standard deviations.

	PB	PG (%)	PI	PB	PG (%)	PI
	MVC-S			MIS-S		
BnB	448.63±9.58	1.32±0.43	66.1±13.1	-2,014.85±20.04	5.10±0.69	222.8±25.9
LB	453.45±11.81	2.35±1.30	102.2±35.9	-2,079.07±14.34	2.07±0.44	130.9±13.6
RANDOM	447.06±11.21	0.96±1.26	38.0±44.8	-2,117.92±11.31	0.24±0.14	22.1±5.0
GRAPH	447.14±10.83	0.98±1.20	42.9±44.0	-2,116.15±11.58	0.32±0.15	31.8±5.0
LB-RELAX	449.01±11.53	1.38±1.51	57.0±51.2	-2,109.17±11.17	0.65±0.20	46.9±6.5
IL-LNS	444.00±9.73	0.29±0.23	19.2±10.2	-2,118.38±11.77	0.22±0.17	19.4±5.8
RL-LNS	445.45±9.99	0.61±0.34	29.6±11.5	-2,118.44±11.36	0.22±0.14	17.2±5.2
CL-LNS	**443.48±9.56**	**0.17±0.09**	**8.7±6.7**	**-2,119.78±12.14**	**0.15±0.15**	**12.8±5.4**
	CA-S			SC-S		
BnB	-113,608±1,611	2.28±0.59	137.4±25.9	171.22±12.50	1.13±0.95	86.7±37.9
LB	-111,342±1,732	4.23±0.75	272.1±26.9	171.39±12.81	1.22±0.97	113.7±35.2
RANDOM	-109,397±1,684	5.90±1.02	235.6±34.9	173.95±12.98	2.67±1.29	124.3±45.4
GRAPH	-108,422±1,775	6.74±1.03	277.7±36.5	185.57±14.17	8.74±2.13	337.8±76.4
LB-RELAX	-114,348±1,516	1.65±0.57	140.5±18.3	170.74±12.35	0.86±0.83	63.2±31.6
IL-LNS	-115,001±1,564	1.09±0.51	90.0±20.8	171.55±12.47	1.33±0.97	63.2±34.3
RL-LNS	-108,920±1,816	6.32±1.03	249.2±35.9	171.14±12.30	1.10±0.77	77.8±28.9
CL-LNS	**-115,513±1,621**	**0.65±0.32**	**50.7±22.7**	**170.11±12.10**	**0.50±0.58**	**26.2±12.8**

Table A.3: Test results on small instances: Primal bound (PB), primal gap (PG) (in percent), primal integral (PI) at 60 minutes time cutoff, averaged over 100 instances and their standard deviations.

	PB	PG (%)	PI	PB	PG (%)	PI
	MVC			MIS		
BnB	919.96±12.38	4.06±0.38	73.4±6.8	-3,888.39±20.62	8.24±0.31	150.5±5.6
LB	900.15±12.32	1.95±0.35	52.6±6.0	-4,009.23±71.94	5.39±1.59	123.1±15.1
RANDOM	886.39±12.71	0.43±0.25	15.6±3.9	-4,225.74±15.63	0.28±0.10	15.8±1.8
GRAPH	886.89±12.79	0.48±0.23	22.9±3.9	-4,206.29±16.76	0.74±0.16	31.6±2.7
LB-RELAX	887.64±12.21	0.57±0.23	39.4±4.4	-4,177.14±18.22	1.42±0.16	48.5±3.0
IL-LNS	885.58±12.65	0.33±0.26	15.9±4.0	-4,216.32±17.30	0.50±0.17	20.4±3.0
RL-LNS	888.89±12.64	0.71±0.30	25.8±4.8	-4,224.37±15.79	0.31±0.13	15.1±2.2
CL-LNS	**883.07±12.61**	**0.05±0.04**	**8.1±2.1**	**-4,226.65±15.56**	**0.26±0.13**	**9.7±2.6**
	CA			SC		
BnB	-216,772±13,060	5.58±5.42	257.1±56.4	109.39±7.26	2.02±1.36	84.4±22.2
LB	-206,526±3,750	10.03±1.39	245.1±19.2	116.43±8.97	7.84±2.88	162.6±39.2
RANDOM	-216,326±2,603	5.76±0.74	129.4±12.1	111.71±7.65	4.02±1.86	100.6±32.0
GRAPH	-213,142±2,713	7.14±0.78	177.6±13.2	112.74±7.64	4.91±1.80	141.7±31.1
LB-RELAX	**-225,154±4,366**	**1.91±1.60**	121.9±23.9	109.26±7.07	1.91±1.42	53.9±24.5
IL-LNS	-214,495±3,148	6.56±1.01	154.0±17.9	109.04±6.94	1.72±1.19	48.1±21.3
RL-LNS	-217,600±2,705	5.20±0.84	106.3±14.2	108.66±6.83	1.38±0.99	98.1±15.1
CL-LNS	-223,257±2,667	2.74±0.71	**95.0±12.5**	**107.78±6.64**	**0.58±0.45**	**28.6±12.6**

Table A.4: Generalization results on large instances: Primal bound (PB), primal gap (PG) (in percent), primal integral (PI) at 30 minutes time cutoff, averaged over 100 instances and their standard deviations.

	PB	PG (%)	PI	PB	PG (%)	PI
	MVC-L			MIS-L		
BnB	904.41±12.95	2.41±0.40	130.2±11.1	-3,970.78±71.54	6.29±1.62	285.1±18.2
LB	893.56±12.62	1.22±0.30	77.8±10.1	-4,079.76±43.09	3.72±0.87	200.7±32.5
RANDOM	886.00±12.74	0.38±0.24	22.7±8.0	**-4,232.68±15.42**	**0.11±0.08**	19.0±3.1
GRAPH	886.34±12.67	0.42±0.23	30.9±7.6	-4,220.89±16.42	0.39±0.15	41.1±5.1
LB-RELAX	886.68±12.33	0.46±0.23	48.4±7.5	-4,199.04±17.54	0.91±0.16	68.6±5.5
IL-LNS	885.00±12.56	0.27±0.23	21.2±8.1	-4,225.28±16.25	0.29±0.15	27.1±5.5
RL-LNS	887.90±12.67	0.59±0.30	37.3±9.6	-4,231.52±15.97	0.14±0.12	18.9±4.1
CL-LNS	**883.07±12.61**	**0.05±0.04**	**9.1±3.4**	-4,232.50±14.86	0.12±0.11	**12.9±4.4**
	CA-L			SC-L		
BnB	-223,225±5,106	2.74±1.87	320.9±83.1	108.87±7.35	1.54±1.33	115.0±42.5
LB	-208,500±3,976	9.17±1.43	414.0±36.9	115.12±8.77	6.80±2.73	293.5±79.7
RANDOM	-217,204±2,612	5.37±0.75	229.2±24.4	110.88±7.55	3.31±1.79	166.4±61.3
GRAPH	-214,926±2,649	6.37±0.86	297.5±26.9	111.49±7.51	3.85±1.74	218.9±56.7
LB-RELAX	-225,848±4,201	1.61±1.50	153.0±50.3	109.26±7.07	1.91±1.42	88.3±48.9
IL-LNS	-219,074±3,278	4.56±0.98	254.2±33.4	109.04±6.94	1.72±1.19	79.1±42.4
RL-LNS	-218,273±2,725	4.91±0.81	197.0±28.5	107.87±6.74	0.66±0.72	116.2±27.1
CL-LNS	**-229,331±2,800**	**0.09±0.10**	**116.1±18.0**	**107.78±6.64**	**0.58±0.45**	**39.2±23.2**

Table A.5: Generalization results on large instances: Primal bound (PB), primal gap (PG) (in percent) and primal integral (PI) at 60 minutes time cutoff, averaged over 100 instances and their standard deviations.
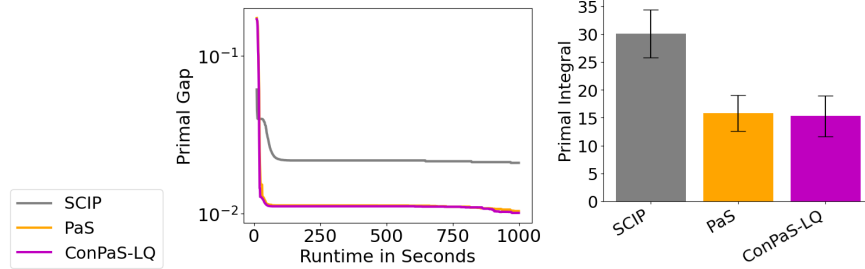
Figure A.6: The primal gap as a function of runtime and the primal integral at 1,000 seconds runtime cutoff. Note that the curves of PaS and ConPaS highly overlap with each other.

and ReLU as the activation function to map them to $\mathbb{R}^{64}$. We then perform two rounds of message passings, the first from variable nodes to constraint nodes and the second from constraint nodes to variable nodes, using graph convolution layers [59] to obtain a final variable embedding. The final variable embedding is then passed through a 2-layer MLP with 64 hidden units per layer and ReLU as the activation function followed by a sigmoid layer to obtain the output $p_{\theta}(x|M)$.

	PaS	ConPaS-Inf	ConPaS-LQ
MVC	$(500, 100, 10)$	$(800, 200, 20)$	$(800, 200, 20)$
MIS	$(600, 600, 5)$	$(1200, 600, 10)$	$(1000, 600, 15)$
CA	$(2000, 0, 0)$	$(2000, 0, 0)$	$(2000, 0, 0)$
IP	$(400, 5, 3)$	$(400, 5, 5)$	$(400, 5, 2)$

Table A.6: Hyperparameters (k_0, k_1, Δ) used for PaS and ConPaS.

A.3.2   Hyperparameter Tuning

In this subsection, we discuss the hyperparameters used for SCIP, ND, PaS and ConPaS.

For SCIP, we fine-tune its restart, presolving and primal heuristic modes on the validation instances. We observe that allowing both restarts and presolving with the aggressive mode turned on for primal heuristics yields the best performance for SCIP. For SCIP with the default mode, it delivers similar primal performance for the CA problem but is worse than the fine-tuned version on others. We also observe that allowing restarts is especially helpful for the IP instances.

	SCIP	ND	PaS	ConPaS-Inf	ConPaS-LQ
MVC	44.5±2.7	10.7±1.2	13.9±6.3	3.1±0.9	**2.8±0.6**
MIS	46.3±2.9	22.9±14.9	34.5±5.8	5.5±1.3	**5.4±1.3**
CA	138.9±28.6	71.0±18.2	28.9±5.6	24.0±6.2	**19.7±4.8**
IP	349.3±87.1	244.0±76.4	236.8±80.6	221.8±73.0	**192.0±67.8**
MVC (large)	88.3±5.0	8.8±2.2	5.0±2.1	3.7±1.1	**2.1±0.8**
CA (large)	167.2±8.2	151.4±21.5	96.9±17.1	39.4±10.4	**28.7±5.7**

Table A.7: Tabular representation of the primal integral plots in Figures 3.10 and 3.11: The primal integral and the standard deviation at 1,000 seconds runtime cutoff averaged over 100 instances.

	PaS	ConPaS-LQ
MVC	$(500, 100, 10)$	$(500, 100, 15)$
MIS	$(500, 500, 10)$	$(500, 500, 10)$
CA	$(1500, 0, 0)$	$(1500, 0, 0)$

Table A.8: Comparisons with Gurobi: Hyperparameters (k_0, k_1, Δ) used for PaS and ConPaS-LQ.

For ND, following [148], we train a model separately for each coverage rate value. Due to limited computing resources, we train models with $\{0.2, 0.3, 0.4\}$ coverage rate values. The best coverage rates we found for the MVC, MIS, CA and IP problems are 0.2, 0.2, 0.4 and 0.3, respectively.

For PaS and ConPaS, the values of k_0, k_1 and Δ are summarized in Table A.6. Note that the best hyperparameters for both MVC and MIS are quite different for PaS and ConPaS. On MVC instances for PaS, we observe that $(k_0, k_1, \Delta) = (600, 200, 20)$ has a smaller primal integral than $(500, 100, 10)$ but has a larger primal gap at 1,000 seconds runtime cutoff. We also test $(k_0, k_1, \Delta) = (500, 100, 10)$ for ConPaS-LQ, it converges to the same primal gaps (with $< 0.002\%$ differences) as $(800, 200, 20)$ but has a 34.1% increase in primal integral. On MIS instances for PaS, we observe that increasing k_0 or Δ (or both) leads to significantly worse performance. However, if we use $(k_0, k_1, \Delta) = (600, 600, 6)$ for ConPaS-LQ, it converges to the same primal gaps (with $< 0.032\%$ differences) as $(1000, 600, 15)$ but has a 131.8% increase in primal integral (still being better than any other baseline).
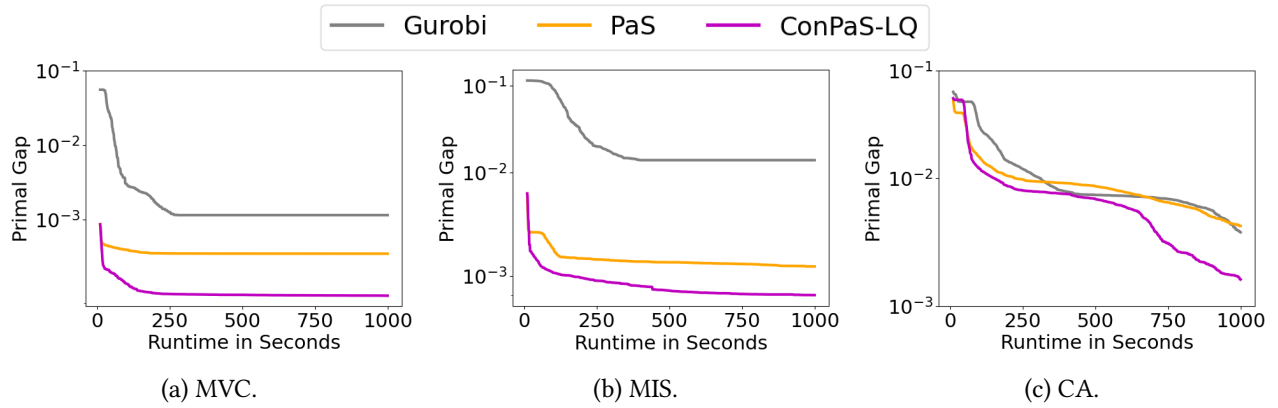
Figure A.7: Comparisons with Gurobi: The primal gap (the lower, the better) as a function of runtime averaged over 100 test instances.
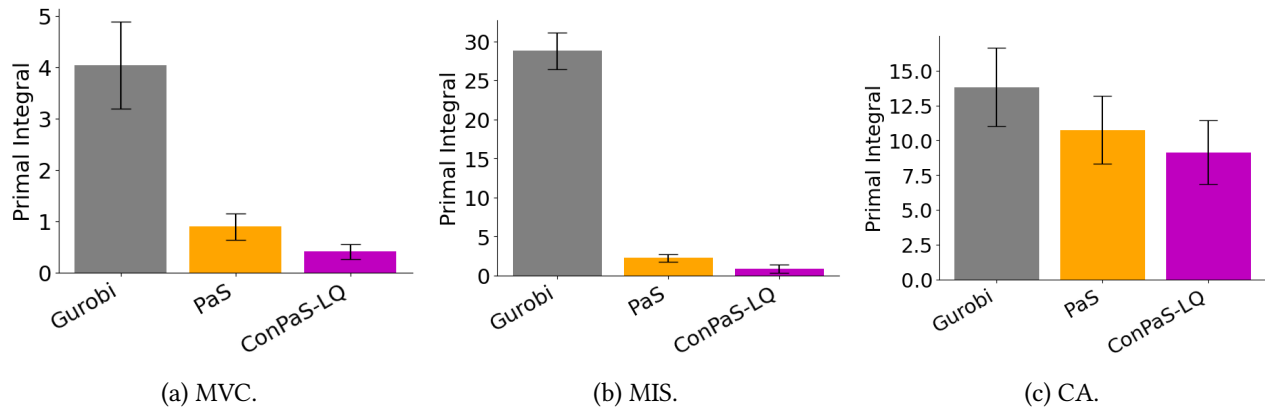

Figure A.8: Comparisons with Gurobi: The primal integral (the lower, the better) at 1,000 seconds runtime cutoff, averaged over 100 test instances. The error bars represent the standard deviation.

	MVC		CA	
	Accuracy	AUROC	Accuracy	AUROC
PaS	81.2%	0.88	88.3%	0.87
ConPaS-LQ	76.9%	0.91	86.9%	0.86

Table A.9: Prediction accuracy and AUROC on 100 validation instances.

A.3.3 Additional Experimental Results

Results on the Workload Appointment Problem   Figure A.6 presents the results on the WA instances. Both PaS and ConPaS-LQ outperform SCIP significantly in terms of the primal gap and the primal integral. However, both approaches converge quickly to low primal gaps, with ConPaS-LQ being very slightly better than PaS.

Comparisons with Gurobi   We compare the performance of ConPaS-LQ against PaS and Gurobi on the MVC, MIS and CA instances. Note that in this experiment, we use Gurobi in the Predict-and-Search phase for both PaS and ConPaS-LQ to ensure a fair comparison. The hyperparameters (k_0, k_1, Δ) are reported in Table A.8. Figure A.7 shows the primal gap as a function of runtime. Figure A.8 shows the primal integral at 1,000 seconds runtime cutoff. The results show that both PaS and ConPaS-LQ outperform Gurobi significantly on MVC and MIS instances. Overall, ConPaS-LQ is still the best when applied on Gurobi.

Prediction Accuracy   To assess how accurate the predicted solutions by the neural networks are, we report the classification accuracy over all binary variables (with the threshold set to 0.5) in Table A.9. We report it for both PaS and ConPaS-LQ on the MVC and CA problems on 100 validation instances. The accuracy is the fraction of correctly classified variables averaged over 50 positive samples for each instance, and we report the average accuracy over 100 validation instances. Since the classification accuracy is sensitive to the threshold, we also report the AUROC. On the MVC instances, though ConPaS has a lower accuracy (w.r.t. the threshold of 0.5), it has higher AUROC than PaS. On the CA instances, their accuracies

and AUROCs are similar. However, we would like to point out that a better accuracy/AUROC does not necessarily indicate a better downstream search performance.