PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Machine Learning Algorithms for Combinatorial Optimization
Auteur: Author:	Defeng Liu
Date:	2022
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Liu, D. (2022). Machine Learning Algorithms for Combinatorial Optimization [Ph.D. thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/10460/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/10460/
Directeurs de recherche: Advisors:	Andrea Lodi
Programme: Program:	Doctorat en mathématiques

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Machine Learning Algorithms for Combinatorial Optimization

DEFENG LIU
Département de mathématiques et de génie industriel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Mathématiques

Août 2022

© Defeng Liu, 2022.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Machine Learning Algorithms for Combinatorial Optimization

présentée par **Defeng LIU**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Louis-Martin ROUSSEAU, président
Andrea LODI, membre et directeur de recherche
Didier CHÉTELAT, membre
Pascal VAN HENTENRYCK, membre externe

DEDICATION

To my parents

ACKNOWLEDGEMENTS

"We are at the very beginning of time for the human race. It is not unreasonable that we grapple with problems. There are tens of thousands of years in the future. Our responsibility is to do what we can, learn what we can, improve the solutions and pass them on. It is our responsibility to leave the men of the future a free hand. [...] It is our responsibility as scientists, knowing the great progress which comes from a satisfactory philosophy of ignorance, the great progress which is the fruit of freedom of thought, to proclaim the value of this freedom, to teach how doubt is not to be feared but welcomed and discussed, and to demand this freedom as our duty to all coming generations."

Richard P. Feynman

This long journey of Ph.D. would not have been possible without the support of my advisor, Andrea Lodi. Thank you for giving me a great freedom of research, while still providing me with valuable advice. Thank you also for helping me get integrated into our incredible DS4DM research chair and for providing opportunities to showcase my work in the community.

The daily struggles of administration and computer networks would have been much harder without the amazing help that the CERC DS4DM team provided. Thank you Mehdi, Khalid, Koladé, Mariia, Pierre for going out of your way to support me.

I am grateful to have found in the DS4DM chair, a considerate and stimulating work environment. Thank you to the whole community, and in particular to Didier, Maxime, Elias, Claudio, Antoine, Giulia, Jaime, Leandro, Federico, Gabriele, Jiaqi, Matteo, Can and Guanyi for the countless and fruitful discussions.

Special thanks go to all the co-authors of the works presented in this thesis: Mathieu Tanneau, Matteo Fischetti, Vincent Perreault, and Alain Hertz, thank you for your trust, patience and commitment. Our collaborations allowed me to grow scientifically and personally, and I have learned a lot from you all. It has been a pleasure to work together!

To my close friends in Montreal and to my family in China, thank you for supporting me with your love and camaraderie.

RÉSUMÉ

Une variété de tâches de décision dans la science et l'industrie modernes sont des problèmes d'*optimisation combinatoire*. Pour résoudre ces problèmes difficiles, des recherches étonnantes ont été menées dans le domaine au cours des dernières décennies. Bien que les méthodes et les techniques d'optimisation développées aient conduit à une multitude d'outils d'optimisation et aient été appliquées avec succès pour résoudre régulièrement un grand nombre de problèmes, il existe encore de nombreux cas pratiques où les techniques existantes ne sont pas adéquates et il est toujours impératif de concevoir et de mettre en œuvre de nouveaux algorithmes et stratégies d'optimisation combinatoire.

L'*apprentissage automatique* est un sous-domaine de l'*intelligence artificielle*. Ces dernières années, l'application des techniques d'apprentissage automatique dans l'optimisation combinatoire est devenue un domaine de recherche émergent. Dans cette thèse, nous visons à renforcer cette direction et nous soutenons que l'apprentissage automatique est un complément prometteur à l'optimisation combinatoire. Plus précisément, nous étudions différents paradigmes pour appliquer l'apprentissage automatique en combinaison avec l'optimisation combinatoire et présentons trois contributions ci-dessous.

Tout d'abord, nous considérons un problème d'optimisation combinatoire spécifique et proposons un cadre pour l'apprentissage d'heuristiques constructives produisant des solutions de haute qualité. Les résultats montrent que notre approche d'apprentissage atteint des performances de généralisation remarquables sur des graphes de plus grande taille et d'une distribution différente.

Deuxièmement, nous considérons l'application de l'apprentissage automatique pour les décisions au sein d'algorithmes d'optimisation combinatoire et proposons un cadre d'apprentissage pour prédire et pour adapter la taille du voisinage de l'heuristique, le *branchement local*, pour la programmation en nombres entiers. Nous montrons informatiquement que les décisions algorithmiques critiques au sein de l'heuristique peuvent en effet être apprises par apprentissage automatique. L'algorithme résultant se généralise bien à la fois au niveau de la taille de l'instance et, remarquablement, entre les instances.

Enfin, nous présentons une contribution méthodologique pour l'intégration de l'apprentissage automatique dans les métaheuristiques pour la résolution de problèmes généraux d'optimisation combinatoire. Plus précisément, nous proposons un cadre général d'apprentissage automatique pour la génération de voisins dans la recherche de métaheuristiques. La clé de la méthodologie proposée réside dans la définition et la génération de voisinages de solutions prometteuses.

Nous proposons un cadre de classification qui exploite les propriétés structurelles du problème et des solutions de haute qualité et qui sélectionne un sous-ensemble de variables pour définir un voisinage de recherche prometteur pour la recherche métaheuristique. Nous démontrons l'efficacité de notre cadre sur deux schémas métaheuristiques, et les résultats expérimentaux indiquent que notre approche permet d'apprendre un compromis satisfaisant entre l'exploration d'un espace de solutions plus large et l'exploitation de régions locales prometteuses.

ABSTRACT

A variety of decision tasks in modern science and industry are Combinatorial Optimization (CO) problems. To solve challenging CO problems, there has been stunning researches in the field over the past decades. Although the optimization methods and techniques developed have led to a vast library of optimization tools and have been successfully applied to routinely solve a large number of CO problems, there are still many practical cases where existing optimization techniques are not adequate and it is always compelling to design and implement new CO algorithms and strategies.

In recent years, the application of Machine Learning (ML) techniques in CO became an emerging research area. In this thesis, we aim to reinforce this direction and we argue that ML is a promising complement to CO. Specifically, we study different paradigms for applying ML in combination with CO and present three contributions.

First, we consider a specific CO problem, the *chordal extension* of a graph, and propose a framework for learning heuristic strategies that yield high-quality solutions. In particular, we propose an imitation learning scheme for learning constructive heuristics. The experimental results demonstrate that our approach achieves remarkable generalization performance on graphs of larger size and from a different distribution.

Second, we consider the application of ML for making decisions within CO algorithms and propose a learning framework for adapting the crucial parameters of the *Local Branching* (LB) heuristic for Mixed-Integer Linear Programming (MILP). The resulting ML strategies are integrated into the LB algorithm and interact with the MILP solver. We computationally show that the critical algorithmic decisions within a CO algorithm can indeed be learned by ML, and the resulting algorithms generalize well both with respect to the instance size and, remarkably, across instances.

Finally, we present a methodological contribution for integrating ML into metaheuristics (MHs) for solving general CO problems. Specifically, we propose a general ML framework for neighbor generation in MH search. The key of the proposed methodology lies in the definition and generation of promising solution neighborhoods. We propose a framework that exploits structural properties from the problem and from its high-quality solutions, and apply the learned knowledge to define promising neighborhoods for MH search. We validate the effectiveness of our framework on two MH schemes: *Tabu Search* and *Large Neighborhood Search*. The experiments show that our approach is able to learn a good trade-off between the exploration of a larger solution space and the exploitation of promising local regions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

AI	Artificial Intelligence
ANN	Artificial Neural Network
B&B	Branch and Bound
B&C	Branch and Cut
CE	Chordal Extension
CG	Chordal Graph
CA	Combinatorial Auction
CO	Combinatorial Optimization
CS	Computer Science
CP	Constraint Programming
CNN	Convolutional Neural Network
DL	Deep Learning
DRL	Deep Reinforcement Learning
DP	Dynamic Programming
GISP	Generalized Independent Set Problem
GNN	Graph Neural Network
GE	Graph Elimination
IID	Independent and Identically Distributed
IL	Imitation Learning
LNS	Large Neighborhood Search
LP	Linear Programming
LB	Local Branching
MIS	Maximum Independent Set
MDP	Markov Decision Process
MH	Metaheuristic
MILP	Mixed-Integer Linear Programming
MIP	Mixed-Integer Programming
MIQP	Mixed-Integer Quadratic Programming
MLP	Multilayer Perceptron
NS	Neighborhood Search
NN	Neural Network
OR	Operations Research
RL	Reinforcement Learning

SDP	Semidefinite Optimization
SL	Supervised Learning
RNN	Recurrent Neural Networks
SSL	Semi-Supervised Learning
SC	Set Covering
TS	Tabu Search
TSP	Traveling Salesman Problem
USL	Unsupervised Learning
WNO	Wireless Network Optimization

CHAPTER 1 INTRODUCTION

1.1 Motivation

In modern science and industry, decision-making tasks arise in the planning and operations of complex systems for a variety of applications, including but not limited to power systems, transportation, logistics and healthcare. These tasks can be formulated as optimization problems, where mathematical models are built and solved to provide operational solutions. In particular, Combinatorial Optimization (CO) is a branch of optimization and generally applied to tackle problems that involve discrete decisions. A CO problem is typically characterized by a set of decision variables with a specified solution space and has the goal of finding an optimal solution with respect to some cost or objective functions.

A large part of CO problems are *NP-hard*, i.e., no polynomial-time algorithm is known to solve them. To solve challenging CO problems, there has been stunning researches in the field over the past decades. Major developments of CO methods and techniques include exact methods, heuristics and metaheuristics.

In the class of exact methods, Mixed-Integer Programming (MIP) is one of the main paradigms for modeling and solving CO problems. The exact resolution of a MIP model is generally attempted by a tree-search framework, where the solution space is split into subspaces and the optimality of the solutions is implicitly checked in the expanded tree. To design efficient MIP algorithms, practical MIP solvers incorporate a variety of complex and efficient algorithmic techniques, such as Branch and Bound (B&B) [1] and *cutting planes* [2]. These techniques have been effectively incorporated in sophisticated software tools [3].

Exact methods are guaranteed to find the optimal solutions as well as a proof of their optimality. On the one hand, due to the NP-hardness nature of many CO problems, solving them to optimality within an exact algorithm is still a very challenging task. On the other hand, in many practical applications, one is more interested in finding a good solution within a reasonable time rather than getting the optimal one. Those practical requirements have motivated the development of *specific heuristics* and *metaheuristics*. Specific heuristics are usually designed for solving a specific type of CO problem; whereas metaheuristics are more general frameworks that provide guidelines and high-level strategies for designing an efficient heuristic algorithm and can be applied to address different types of CO problems.

Those methods and techniques developed in the past have led to a vast library of optimization tools and have been successfully applied to solve a large number of CO problems. This

progress has stimulated the community to tackle more difficult large-scale CO problems, and some of them are still very difficult to solve. Nonetheless, there are still many practical cases where existing optimization techniques are not adequate and it is always compelling to design and implement new CO algorithms.

From a methodological point of view, all the CO algorithms can be viewed as conducting a series of search processes in the solution space, either exact, where the entire solution space is "conquered" (reminding the "divide-and-conquer" framework) by (implicit) enumeration, or heuristic, where typically only some part of the solution space is explored. It is worth noting that, a lot of information is produced from these processes, and therefore, a large volume of data can be collected. These data might provide valuable information about the optimization status of the process, the characteristics of the problem, the structures and properties of high-quality solutions in the search regions being visited. However, such knowledge has not been fully exploited by the traditional CO algorithms.

From an application point of view, instances in many real-world applications are solved repeatedly. These instances are similar to each other in terms of the structure of the problem. When classical CO algorithms are called, they usually solve each of these instances from scratch. For some large-size applications, this can take a long computation time. In many practical cases, high-quality solutions are often required within a very restricted time frame, and there might not be adequate computation resources to do so. Hence, repeating complex optimization processes to solve similar instances is an issue.

Conversely, in Artificial Intelligence (AI), the concept of Machine Learning (ML) is well rooted as a principle underlying the development of intelligent systems for knowledge extraction and decision-making by "learning from data". In a nutshell, ML has the goal of extracting patterns (or knowledge) from observed data in the past, and making predictions on new data. In recent years, there has been stimulating progress in ML techniques including Supervised Learning (SL), Semi-Supervised Learning (SSL), Unsupervised Learning (USL), Reinforcement Learning (RL) and Artificial neural Networks (ANN). resulting in impressive achievements in natural language processing, computer vision and recommendation systems. These achievements have motivated emerging research interest in the application of ML for designing novel CO algorithms.

As discussed before, with more and more similar instances solved and a huge volume of data collected from the optimization processes, frequencies and patterns appear. This provides a great opportunity for ML to extract useful knowledge from data and design "learning-based" CO algorithms.

This thesis aims to reinforce this research direction, in which ML techniques can be leveraged

for CO. Specifically, we investigate different paradigms for applying ML in combination with CO, and present three contributions. The remainder of this chapter introduces the necessary background and notation on CO and ML.

1.2 Background

1.2.1 Combinatorial Optimization

CO is a branch of optimization and a CO problem is generally defined by a set of variables, a constrained solution space and some objective function(s). The solution space can be specified by constraints in any form. There are different mathematical paradigms for modeling and solving CO problems, including MIP, Constraint Programming (CP), and Dynamic Programming (DP). Without loss of generality, a CO problem can be formulated into a MIP model as follows:

$$
\begin{aligned}
(P0) \quad \min \quad & c(\boldsymbol{x}) && (1.1)\\
\text{s.t.} \quad & g(\boldsymbol{x}) \leq \boldsymbol{b}, && (1.2)\\
& x_i \in \mathbb{Z}^+,\ \forall i \in \mathcal{I}, \quad x_j \geq 0,\ \forall j \in \mathcal{C}, && (1.3)
\end{aligned}
$$

where the index set $\mathcal{V} := \{1, \ldots, n\}$ of decision variables is partitioned into $\mathcal{I}, \mathcal{C}$, which are the index sets of integer and continuous variables, respectively.

In particular, when the objective function and the constraints are all linear, the problem can be formulated into a Mixed-Integer Linear Programming (MILP) model, namely

$$
\begin{aligned}
(P1) \quad \min \quad & \boldsymbol{c}^T\boldsymbol{x} && (1.4)\\
\text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, && (1.5)\\
& x_i \in \mathbb{Z}^+,\ \forall i \in \mathcal{I}, \quad x_j \geq 0,\ \forall j \in \mathcal{C}. && (1.6)
\end{aligned}
$$

MILP has been the workhorse for solving MIP problems in modern MIP solvers. The solution process of a MILP problem is generally attempted by a B&B scheme, where a B&B tree is generated to split the solution space into subspaces, and each node in the tree represents a subproblem. At each node in the tree, a Linear Programming (LP) problem – the continuous relaxation of the subproblem – is solved to optimality and the feasibility of the LP solution is checked in the original MILP problem. If the LP solution satisfies $x_j \in \mathbb{Z}^+,\ \forall j \in \mathcal{I}$, then the node is a leaf node and the LP is also a feasible solution of the original MILP; otherwise, an integer variable with fractional values is selected to generate two branching nodes from the

current one. In the cases where the LP solution of a node has a larger objective than the best *incumbent* solution of the original MILP, the node will be pruned from the tree and no branching will be processed from it. An efficient B&B algorithm can reduce the size of the B&B tree – therefore avoiding the enumeration of the entire solution space – by pruning a large number of nodes. Modern MILP solvers have been integrated with a variety of efficient algorithmic components, including cutting planes, primal heuristics [4], pre-processing [5] and auxiliary searching rules [6–8].

Although modern MIP solvers have achieved a stunning progress and have been applied to solve CO problems in various applications, there are still many practical cases where the CO problems are hard and complex, and the attention is focused on finding good-quality solutions with a short computing time rather than solving the problem to proven optimality. Therefore, efficient approximate methods such as heuristics or metaheuristics are developed to compute fast solutions. This thesis presents several ML-based methods to improve heuristic and metaheuristic algorithms.

1.2.2 Machine Learning

In ML, the goal is to detect patterns from a set of observed data and make predictions about future data. We can formalize a standard ML prediction problem as follows. Given a variable space $\mathcal{Z}$ and a set of examples $\mathcal{D}_{\mathcal{Z}} = \{z_1, z_2, \ldots, z_m\}$ from the unknown distribution $\mathcal{P}(\mathcal{Z})$, the task is to find a function f over a family of functions $\mathcal{F}$, such that f "fits" well on $\mathcal{P}(\mathcal{Z})$. It is assumed that all the observed examples are drawn Independent and Identically Distributed (IID) from the same distribution $\mathcal{P}(\mathcal{Z})$.

If a *loss* function $\mathcal{L} : \mathcal{F} \times \mathcal{Z} \to \mathbb{R}$ is specified to measure the performance of f and a set of examples $\mathcal{D}_{\mathcal{Z}}$ sampled from $\mathcal{P}(\mathcal{Z})$ is available, a ML task can be defined as finding $\hat{f} \in \mathcal{F}$ that minimizes the *empirical loss* on $\mathcal{D}_{\mathcal{Z}}$, i.e.,

$$\hat{f} = \operatorname*{argmin}_{f \in \mathcal{F}} \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f, z_i). \tag{1.7}$$

The differences in forms and contents of $\mathcal{Z}$, $\mathcal{F}$, $\mathcal{L}$ result in different learning tasks.

Supervised Learning In SL, the variable space $\mathcal{Z}$ consists of $\mathcal{X} \times \mathcal{Y}$, where $\mathcal{X}$ is the space of input variables and $\mathcal{Y}$ is the space of output variables. The family of functions $\mathcal{F}$ is a set of mappings $f : \mathcal{X} \mapsto \mathcal{Y}$. For any sample $(x, y) \in \mathcal{X} \times \mathcal{Y}$, the loss function $\mathcal{L}$ measures the discrepancy between $f(x)$ and y. Ideally, the output $\mathcal{Y}$ can be in any form. However, most tasks assume that $\mathcal{Y}$ is categorical or numerical. The former characterizes the task as

classification, whereas the latter induces *regression*.

Reinforcement Learning When the decision process can be modeled by a Markov Decision Process (MDP) and one is interested in extract knowledge from experience, RL can be employed to learn how to make decisions with the environment in an interactive way. The goal of RL is to retrieve the optimal decision policy that maximizes the expected return with respect to the task based on the *Bellman's principle* [9]. RL methods can be roughly divided into *value-based* and *policy-based* methods. Value-based methods learn policies from the estimate of action-value functions, whereas policy-based methods learn a parameterized policy directly without the estimation of value functions. Policy-based methods are generally expected to have good potential performances for CO problems, because, in many CO applications, learning a policy to directly make decisions is more straightforward than estimating some expected values of a policy [10].

CHAPTER 2 CRITICAL LITERATURE REVIEW

ML techniques were applied to solve CO problems for the first time in 1980's. The survey [11] introduced early works on using Hopfield networks and self-organizing networks to solve a variety of problems, including assignment problems, clustering problems, packing, scheduling and graph problems. However, due to restricted computational power and hardness of training, advancements in this area were sparse at the beginning of this century.

The recent progress in ML, such as Deep Learning (DL) [12, 13] and RL [10], has led to inspiring results in natural language processing, computer vision and chess playing [14]. These successes have stimulated renewed research interest in learning algorithms for CO problems and have generated a variety of works in the literature recently surveyed in [15–18]. The remainder of the chapter first gives a short overview of the different paradigms that leverage ML techniques to solve CO problems and then schematically reviews the ML methods related to the present thesis.

2.1 Paradigms of combining ML with CO

There are three main paradigms that have been used to leverage ML for CO.

2.1.1 End-to-end learning

One of the first paradigms in the literature is *end-to-end learning*. This approach is to learn heuristics by using ML to generate solutions to CO problems directly. The Traveling Salesman Problem (TSP) is an example of a frequently studied problem and has received recurring interest in the literature. TSP is a NP-hard CO problem, where given a graph, one needs to search the space of permutations to find an optimal sequence of nodes to visit with minimal total edge weights (tour length). Additionally, many heuristic algorithms have been developed to build practical solvers for this problem. As a result, TSP is a good benchmark problem, with vast baseline algorithms, to test any new CO algorithm.

Pointer networks [19] were introduced as a type of Recurrent Neural Networks (RNN), wherein an encoder-decoder architecture embeds the node set and produces a permutation of nodes to construct a solution. The *pointer networks* are used as policy approximators and are trained via SL with labels provided by a TSP solver. A drawback of the method in [19] is that the labels are not always accessible in practical applications. To address that issue, a RL approach to learn policies from experience was proposed in [20]. The authors formulated the solution

procedure as a trivial MDP, in which the only non-terminal state is the input graph. The policy networks are trained through policy gradient methods. The authors' experiments on TSP and knapsack problems showed positive results on instances of the same size. Later, the work of [21] applied graph embeddings networks and RL to heuristically solve CO problems on graphs. In [22], the use of an attention model based on graph attention networks [23] and the transformer architecture [24] was proposed. One distinct feature of a graph embedded model is that it is invariant to the ordering of input and it can naturally be applied to general problems on graphs.

2.1.2 Learning algorithmic strategies

Unlike end-to-end learning, which generates solutions for a given problem directly using ML, more recent works that integrate ML into existing CO algorithms have gained increasing attention in the field. These approaches investigated the use of ML to learn strategies for making algorithmic decisions taken inside CO algorithms. The motivation can be either to learn a cheaper approximation of a computationally expensive algorithmic strategy or to explore new strategies that make promising decisions for applications where expert knowledge is inadequate.

For instance, although designed and implemented as an exact framework for solving MIP problems, modern MIP solvers have incorporated a collection of algorithmic strategies and many of these ideas are inherently heuristic. Therefore, it is not surprising that MIP algorithms can potentially benefit from data and learning. Nonetheless, the application of ML for improving MIP solvers have received emerging research interest. For Branch and Cut (B&C) related decision strategies, the works of [25–27] proposed an imitation learning framework to train ML policies to approximate existing branching heuristics for selecting branching variables. The work of [28] also applied an imitation approach to learn a node search strategy for node selection. For cutting plane selection, the authors of [29] proposed a ML-based framework for selecting cutting planes adaptively through a RL formulation. ML approaches for other auxiliary tasks within a MIP solver have also been investigated: learning strategies for executing primal heuristics [30, 31]; learning scaling strategies for reducing numerical errors and for improving the dual simplex algorithm [32]; learning to decide on the linearization of MIQPs [33]; hyperparameter configuration [34].

2.1.3 Extraction of valuable information

The objective of these approaches is to learn valuable information from a particular problem context or from an optimization process through ML. Although the extracted knowledge itself

does not directly improve the underlying CO algorithm, it offers analysis of the characteristics of CO problem and the patterns of the optimization algorithm. These insights can be potentially exploited for designing new optimization strategies by human experts or by automated decision systems.

The target information can be an evaluation of a given problem or a CO algorithm, such as the difficulty of the problem and the outcome of an optimization process. For instance, the authors of [35] and [36] used RL methods to learn an evaluation function for a given heuristic. The learned value function can be then applied in replacement of the original objective to enhance heuristic search. The value function can also be used as a function approximator to predict the objective of a CO model. In the context of cutting plane methods in MIP, the work of [37] trained a neural network as a regression model to approximate the objective of sub-problems generated by adding candidate cuts. The learned information can be applied to design algorithms for selecting the most promising cuts based on the predicted improvement of lower bound. Also, in [38] the authors leveraged deep learning to predict tactical solutions to an operational planning problem. Morevoer, in [39], a classification framework was proposed to evaluate the resolution outcome of a general MILP. The input features of the MILP resolution progress are sequentially collected after a fraction of the specified total time. As a result, a binary decision is made to predict whether the problem can be solved before timing out.

2.2 Learning methods

The previous section highlighted some important use cases and algorithmic structures where ML strategies can be combined with CO. In this section, we present a methodological review on the practical learning methods for building and training those ML models.

2.2.1 Representation

In DL, Neural Networks (NN) have been developed and applied to tackle various problems in ML. As anticipated, early research on applying Hopfield networks and self-organizing networks to solve CO problems was first addressed in the 1980's and 1990's [11]. Recently, different types of NN methods, based on feedforward neural networks, have been leveraged to build predictive models and to process a variety of optimization data from CO problems.

The work of [38] applied a Multilayer Perceptron (MLP) to encode the context of a tactical planning problem in intermodal railway transportation. An alternative type of feedforward neural network, known as RNN, is suitable for processing sequence data. The pointer networks

introduced in [19] were used to sequentially encode the nodes of TSP and to predict solutions in the output layer. The work [21] applied graph embeddings and NNs to process data from network problems such as minimum vertex cover, maximum cut and TSP. Moreover, Graph Neural Networks (GNN) are an expressive type of model to encode data with a graph format by propagating information from the neighborhood [40]. Due to their appealing scalability and flexibility, GNNs have become an emerging class of NNs for building ML models for various CO tasks [18, 27, 41]. The Convolutional Neural Networks (CNN) are wildly used to process image data in Computer Vision (CV). Although CNNs are still rarely used to tackle data of CO problems, it is worth noting that CNN models have been successfully applied in a ML-based framework to master chess games [14, 42].

2.2.2 Training

In the literature, there are roughly two branches of learning methods applied to train ML models for solving CO problems, SL and RL. For the applications where one has empirical knowledge about the values or policies to be learned, i.e., labels of good decisions are accessible, SL can be applied to learn from labels or demonstrations. In other applications, where knowledge is not adequate to solve the problem, it is desirable to explore new knowledge or decision strategies. In these cases, RL can be applied to learn from experience.

A detailed survey on the two types of learning mechanisms for CO has been compiled in [16]. Here, we just highlight the fact that in the context of learning ML models, SL and RL are not mutually exclusive. In some applications, ML models can be trained by combining both approaches. If the objective is to learn new decision strategies that improve the state of the art, the ML model can be initially trained through a SL approach such as imitation learning to achieve state-of-the-art performance. Then, the pretrained policy can be further improved by using RL. For instance, in the context of branching strategies for B&B algorithms, strong branching is an effective heuristic but computationally expensive. Then, the works of [25, 27] proposed to use imitation learning to approximate expert branching strategies. We believe that policies learned from strong branching have the potential to be improved by RL. In another example [14], the authors trained ML models to play the game of Go within a learning-based framework in the combination of SL and RL.

CHAPTER 3 DISCUSSION OF THE RESEARCH PROJECT AS A WHOLE AND GENERAL ORGANIZATION OF THE DISSERTATION

This thesis aims to reinforce the research direction in which ML techniques can be leveraged for CO. Specifically, we study different paradigms for applying ML in combination with CO, and see ML as a lens for exploring valuable knowledge and patterns that classical CO algorithms are not aware of. We aim to show that the use of ML can indeed produce efficient algorithmic strategies for CO by exploiting a deeper understanding of the relation between the critical characteristics of the problem and its solutions, and the actual behavior of CO algorithms.

3.1 Contributions

Learning chordal extensions (Chapter 4)

In our first work, presented in Chapter 4, we consider the computation of *chordal extensions*, a specific CO problem with a variety of applications in numerical optimization, and propose a framework for learning heuristic strategies yielding high-quality solutions for the problem. As a first building block of the learning framework, we propose an imitation learning scheme for learning constructive heuristics. Our ML models are trained from the graph-structured data of the problem, and the resulting ML strategy is sequentially called to construct a solution. The results show that our learning-based approach achieves remarkable generalization performance on graphs of larger size and from a different distribution. Another desirable behavior of our approach is that it allows to speed up the learning process by training on smaller synthetic problems with a marginal loss of performance.

Revisiting local branching with a machine learning lens (Chapter 5)

The initial work has led us to consider the application of ML for solving general CO problems. Our second contribution, presented in Chapter 5, addresses the algorithmic decisions within MILP. In particular, we focus on Local Branching (LB), a well-known primal heuristic and derive a learning framework for predicting and adapting its neighborhood size. The developed ML models learn from not only the properties of the problem and its solutions, but also from the information collected during the optimization process. The resulting ML strategies are integrated into the LB algorithm and interact with the solver at each iteration. We computationally show that the critical algorithmic decisions within a CO algorithm can indeed be learned by ML and the resulting algorithms generalize well both with respect to

the instance size and, remarkably, across instances.

Learning to generate neighbors in metaheuristic search (Chapter 6)

Finally, in Chapter 6, we present a methodological contribution for integrating ML into metaheuristics for solving general CO problems. Specifically, we propose a general ML framework for neighbor generation in metaheuristic search. The key of the proposed methodology lies in the definition and generation of promising solution neighborhoods. The developed classification framework exploits structural properties both from the problem and from its high-quality solutions, and selects a subset of variables to define a promising search neighborhood for metaheuristic search. We demonstrate the effectiveness of our framework on two metaheuristic schemes: *Tabu Search* and *Large Neighborhood Search*. The experimental results indicate that our approach is able to learn a satisfactory trade-off between the exploration of a larger solution space and the exploitation of promising local regions.

3.2 Outline of the thesis

The remainder of this document is organized as follows. Chapters 4, 5 and 6 form the main body of this thesis, and contain the three contributions outlined above. Chapter 7 discusses and highlights the connections among the three contributions. Finally, Chapter 8 presents a summary of this thesis, comments on its limitations, and discusses future research directions.

CHAPTER 4 ARTICLE 1: LEARNING CHORDAL EXTENSIONS

Authors: Defeng Liu, Andrea Lodi, Mathieu Tanneau
Published *Journal of Global Optimization* [43]. Date: 04 January, 2021.

Abstract A highly influential ingredient of many techniques designed to exploit sparsity in numerical optimization is the so-called chordal extension of a graph representation of the optimization problem. The definitive relation between chordal extension and the performance of the optimization algorithm that uses the extension is not a mathematically understood task.

For this reason, we follow the current research trend of looking at Combinatorial Optimization tasks by using a Machine Learning lens, and we devise a framework for learning elimination rules yielding high-quality chordal extensions. As a first building block of the learning framework, we propose an imitation learning scheme that mimics the elimination ordering provided by an expert rule.

Results show that our imitation learning approach is effective in learning two classical elimination rules: the minimum degree and minimum fill-in heuristics, using simple Graph Neural Network models with only a handful of parameters. Moreover, the learned policies display remarkable generalization performance, across both graphs of larger size, and graphs from a different distribution.

4.1 Introduction

A simple undirected graph $G = (V, E)$ is *chordal* if, for every cycle c of length at least four, there exists an edge $e \in E$ that connects two non-consecutive vertices of c. A *chordal extension* of a graph G is a chordal graph H such that G is a sub-graph of H, i.e., one can obtain H by adding edges to G. A practical way of constructing chordal extensions is via *graph elimination* [44], which consists in sequentially eliminating the nodes of the graph. At each step, a node v is selected, new edges are inserted so as to make the neighbors of v into a clique, then v is removed (i.e., eliminated). This process is repeated until all nodes have been eliminated, and one obtains a chordal extension by adding to the original graph all edges that were inserted in the process. The order in which nodes were eliminated is thereby called an *elimination ordering*.

This work focuses on the role of chordal extensions and graph elimination within optimization

frameworks. Indeed, there is a direct connection between chordal extensions, which are typically computed via graph elimination, and a number of classical sparsity-exploiting techniques [44, 45]. In particular, we seek to devise a framework for *learning* elimination rules that yield high-quality chordal extensions, as we illustrate below.

Our first motivating example is the computation of a fill-reducing ordering for sparse Cholesky factorization, a process that reduces to computing an elimination ordering [44, 46]. Crucially, Cholesky factorization underlies most implementations of interior-point algorithms for linear programming [45, 47], and the choice of ordering can have a major impact on the method's performance [48]. Similarly, chordal graphs form the basis of *chordal decomposition* techniques to exploit sparsity in semi-definite programming (SDP) problems, see, e.g., [49–52]. Specifically, a single, dense, semi-definite constraint, can be decomposed into several smaller, yet coupled, semi-definite constraints. This reformulation also reduces to computing a chordal extension, and can dramatically improve the performance of both interior-point and first-order methods on large problems [52, 53]. More generally, a similar approach can be leveraged in linear conic optimization and convex optimization, see, e.g., [44] and [54].

Historically, efforts have focused on computing minimum chordal extensions, i.e., chordal extensions with a minimum number of additional edges [48, 55], which has been proven to be NP-complete [56]. This fostered the development of fast and efficient heuristics such as minimum degree [57] and nested dissection [58] orderings. State-of-the-art implementations of these methods are routinely used in most optimization and sparse linear algebra software, where they tackle problems with up to millions of variables. Nevertheless, depending on the application, lower fill-in may not necessarily translate into improved performance. For instance, in the context of sparse matrix factorization, memory requirements and ability to exploit parallelism are impacted not only by the number of non-zero entries in the factor, i.e., fill-in, but also by the shape of the corresponding elimination tree, as shown in [59, 60]. In addition, the ordering's impact on numerical stability is studied in [61], wherein authors consider a so-called *priority minimum-degree* ordering, which tends to produce more fill-in but is numerically more stable. Furthermore, practical experience with chordal decomposition for sparse SDP problems, as evidenced in [62, 63], suggests that the number and size of the cliques in a chordal extension are more relevant than the number of additional edges. In fact, in [63], the authors state that "*the number of added edges is not always the best criterion to minimize, which proves that the computation of the chordal extension is a question that deserves to be investigated.*" This motivates investigating a broader paradigm for computing chordal extensions.

Recently, the use of *Machine Learning* (ML) in *Combinatorial Optimization* (CO) became

a popular research area with quite a number of contributions investigating many angles of such a connection. On the one hand, some research has been devoted to solve CO problems by ML, i.e., to devise new heuristic algorithms that perform the end-to-end learning of the solution of a CO problem. On the other hand, ML has been used to tackle some tasks within CO algorithms and software for which modern statistical learning has chances to improve the current performances, either because the known way of performing those tasks is computationally heavy or because they are poorly understood from the mathematical standpoint. The interested reader is referred to [16] for a methodological survey on this new research area.

Our work does not follow the first direction outlined in the previous paragraph. Indeed, the goal of this paper is *not* to compete with existing state-of-the-art heuristics for graph elimination. First, these heuristics leverage decades of development and clever engineering, and have been optimized for fast runtime and good solution quality. Second, and more importantly, this work is a first step towards a deeper understanding of the relation between the characteristics of a chordal extension and the behavior of optimization algorithms and software, a topic of interest in its own right and whose mathematical knowledge is currently insufficient. We seek to address this lack of mathematical clarity by leveraging ML-based techniques to *learn* elimination rules from experience. As a byproduct, a better understanding of what "good" chordal extensions look like (for a specific task) can lead to an easier and more direct customization of elimination orderings to sets of similar graphs. On the one hand, the definition of a "high-quality" chordal extension is application specific; in optimization contexts, it may encompass a combination of memory requirements, computing time and numerical accuracy. On the other hand, suitable model priors and learning algorithms are more easily shared between applications. Thus, our present contributions are 1) to propose a mathematical framework for the problem of learning elimination orderings, and 2) to provide methodological and practical insights on the learning process itself. The proposed approach is illustrated in the classical setting of minimum chordal extensions.

The rest of the paper is organized as follows. In Section 4.2, we introduce some relevant definitions and concepts. In Section 4.3, we present our methodology for learning chordal extensions. In Section 4.4, we report on numerical experiments. Section 4.5 gives further discussion and Section 4.6 concludes the paper.

4.2 Basic notations and concepts

In this section, we review some notations and concepts used in the remainder of the paper. Section 4.2.1 introduces basic notations and definitions of graphs and Section 4.2.2 introduces

a commonly used model for sequential decision problems. In Sections 4.2.3 and 4.2.4, we briefly go through some Machine Learning concepts, in order to help the reader be familiar with relevant ML methods and properly locate the methodology we propose in Section 4.3.

In all that follows, for an arbitrary set S, we use the notation $\mathfrak{P}(S)$ to denote the set of all probability distributions over S. In addition, we write $\mathbb{E}_{X\sim\mathcal{D}}[\cdot]$ to indicate that the expectation is computed with respect to random variable X being sampled from distribution $\mathcal{D}$.

4.2.1 Graph-theoretic notations

In this paper, all considered graphs are simple, undirected graphs. A graph is denoted by $G = (V, E)$, where V (resp. E) denotes the set of its nodes (resp. its edges). For an edge $e = (v, w)$, we say that e is *incident* to v and w, that v, w are the *extremities* of e, and that v, w are *adjacent*. The neighborhood of v, denoted by $\mathcal{N}_G(v)$, is defined as the set of nodes that are adjacent to v.

The degree of node v in G is denoted by $\delta_G(v)$. We say that node v is of *minimum degree* if $\delta_G(v) \leq \delta_G(w)$ holds for all nodes $w \in V$. Note that several nodes of minimum degree may exist.

In what follows, we will drop the subscript G whenever context is sufficiently clear, and write for example $\delta(v)$ rather than $\delta_G(v)$.

4.2.2 Markov Decision Processes

Markov Decision Processes (MDPs) [64] are wildly used to formulate sequential decision problems. An MDP is characterized by a set of possible *states* $\mathcal{S}$, a set of possible actions $\mathcal{A}$, the dynamics of the system, and a cost function.

Formally, an MDP is defined by a tuple $(\mathcal{S}, \mathcal{A}, P, c)$, where P encodes the system's dynamics, and c encodes the cost function. For any state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$, $P(s, a)$ is a probability distribution over the state space $\mathcal{S}$. That is, if one takes action a in state s, the next observed state s' will be sampled from the distribution $P(s, a) \in \mathfrak{P}(\mathcal{S})$. Finally, the cost of executing action a in state s is denoted by $c(s, a)$.

Given an MDP $(\mathcal{S}, \mathcal{A}, P, c)$, a *policy* is a decision rule for selecting an action a, given a current state s. Specifically, a policy π is a function

$$\pi : \mathcal{S} \longmapsto \mathfrak{P}(\mathcal{A}) \tag{4.1}$$

$$s \longrightarrow \pi(s), \tag{4.2}$$

and the next action a is sampled from $\pi(s) \in \mathfrak{P}(\mathcal{A})$. If a policy π maps each state to a *single* action, i.e., if $\pi(s)$ is a degenerate distribution for every $s \in \mathcal{S}$, then π is called a *deterministic policy*; otherwise it is called a *stochastic policy*. For the remainder of this paper, we will only consider stochastic policies, and we define the expected *immediate cost* for policy π in state $s \in \mathcal{S}$ as

$$C_\pi(s) = \mathbb{E}_{a\sim\pi(s)}\left[c(s,a)\right]. \tag{4.3}$$

A *trajectory* is a sequence of state-action pairs $\Big((s_0,a_0),(s_1,a_1),...\Big)$ where s_{t+1} is sampled from $P(s_t,a_t)$. In this paper, all trajectories will always be finite, although they may be of arbitrary length. We say that a trajectory is sampled from a policy π if each action a_t is sampled from $\pi(s_t)$. The total cost along a trajectory is then given by

$$\sum_{t\geq 0} c(s_t,a_t), \tag{4.4}$$

which is always finite since we only consider finite trajectories.

Finally, for a distribution of initial states $\mathcal{D}_0 \in \mathfrak{P}(\mathcal{S})$, we define the *expected total cost* of a policy π as

$$C_\pi^{tot} = \mathbb{E}_{s_0\sim\mathcal{D}_0}\left(\mathbb{E}_{a_t\sim\pi(s_t),s_{t+1}\sim P(s_t,a_t)}\left[\sum_{t\geq 0} c(s_t,a_t)\right]\right). \tag{4.5}$$

Furthermore, $\mathcal{D}_0$ and π induce a stationary distribution over states, which we denote by $\mathcal{D}_\pi$. Thus, we define the *expected average cost* of policy π as

$$C_\pi^{avg} = \mathbb{E}_{s\sim\mathcal{D}_\pi}\left[\mathbb{E}_{a\sim\pi(s)}\left(c(s,a)\right)\right] \tag{4.6}$$

$$= \mathbb{E}_{s\sim\mathcal{D}_\pi}\left[C_\pi(s)\right]. \tag{4.7}$$

In this work, we assume that the expectations (4.5) and (4.6) are finite; this ensures that the learning problems defined in Section 4.2.4 are well-defined. Note that this assumption is trivially satisfied if (i) the cost function only takes finite values and (ii) the set of all possible state-action pairs is finite. For ease of reading, we also drop the explicit dependency of C_π^{tot} and C_π^{avg} on $\mathcal{D}_0$, since the latter will always be evident from the context.

4.2.3 Standard statistical learning

In statistical learning, the goal is to detect patterns from a set of observed data and make predictions about future data. We can formalize the standard statistical learning problem as follows. Given a variable space $\mathcal{Z}$ and a set of examples $\mathcal{D}_{\mathcal{Z}} = \{z_1, z_2, \ldots, z_m\}$ from the unknown distribution $\mathcal{P}(\mathcal{Z})$, the task is to find a function f over a family of functions $\mathcal{F}$, such that f "performs well" on $\mathcal{P}(\mathcal{Z})$. It is assumed that all the observed examples are drawn independent and identically distributed (i.i.d.) from the same distribution $\mathcal{P}(\mathcal{Z})$.

If a *loss* function $\mathcal{L} : \mathcal{F} \times \mathcal{Z} \mapsto \mathbb{R}$ is specified to measure the performance of f, then the goal can be described as finding $\hat{f} \in \mathcal{F}$ that minimizes the *expected loss* with respect to $\mathcal{P}(\mathcal{Z})$, i.e.,

$$\hat{f} = \operatorname*{argmin}_{f \in \mathcal{F}} \mathbb{E}_{z \sim \mathcal{P}(\mathcal{Z})}[\mathcal{L}(f, z)]. \tag{4.8}$$

The learned function $\hat{f}$ is then used to predict future data.

However, the expected loss cannot be computed exactly due to the fact that $\mathcal{P}(\mathcal{Z})$ is unknown. In practice, if a subset of examples $\mathcal{D}_{\mathcal{Z}}$ sampled from $\mathcal{P}(\mathcal{Z})$ is available, a number of learning methods turn to minimize the *empirical loss* on $\mathcal{D}_{\mathcal{Z}}$. Then, $\hat{f}$ is obtained by solving

$$\hat{f} = \operatorname*{argmin}_{f \in \mathcal{F}} \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f, z_i). \tag{4.9}$$

The differences in forms and contents of $\mathcal{Z}$, $\mathcal{F}$, $\mathcal{L}$ result in different learning tasks. Here, we only introduce *supervised learning* that is the relevant task for the current state of our work.

Supervised learning. In supervised learning, the variable space $\mathcal{Z}$ consists of $\mathcal{X} \times \mathcal{Y}$, where $\mathcal{X}$ is the space of input variables and $\mathcal{Y}$ is the space of output variables. The family of functions $\mathcal{F}$ is a set of mappings $f : \mathcal{X} \mapsto \mathcal{Y}$. For any sample $(x, y) \in \mathcal{X} \times \mathcal{Y}$, the loss function $\mathcal{L}$ measures the discrepancy between $f(x)$ and y. Ideally, the output $\mathcal{Y}$ can be in any form or intent. However, most tasks assume that $\mathcal{Y}$ is categorical or nominal. The former characterizes the task as classification, whereas the latter induces regression.

4.2.4 Imitation learning for sequential decision problems

Imitation learning (IL) [65–68] is an extension of supervised learning from problems satisfying i.i.d. assumption to sequential decision problems, see, e.g., [69–71].

In IL, the target policy learns its decision rule from an expert policy. More precisely, given the class of candidate policies Π, we seek to find a target policy $\pi \in \Pi$ that matches the expert

policy π^*. The target and expert policy are often referred as the *learner* and the *expert*. The cost is defined by a loss function $\mathcal{L}(\pi, \pi^*)$, a measure of discrepancy between π and π^*. If a *behavior policy* π' is set to generate trajectories of states, the goal is to find a policy $\hat{\pi}$ that minimizes the expected loss with respect to the distribution of states induced by π', namely

$$\hat{\pi} = \underset{\pi \in \Pi}{\operatorname{argmin}} \, \mathbb{E}_{s \sim \mathcal{D}_{\pi'}} [\mathcal{L}(\pi(s), \pi^*(s))]. \tag{4.10}$$

In the literature, various strategies have been employed in terms of the choice of behavior policy. A classic family of IL methods is the supervised approach for imitation learning, which fixes the behavior policy to be the expert. Other methods such as Data Aggregation (DAGGER) [67] propose to use more interactive strategies by generating training data from executing the learner itself.

Supervised imitation learning. The supervised approach for IL fixes the behavior policy as the expert, i.e., $\pi' = \pi^*$. Then the learner is trained under the distribution induced by the expert, given by

$$\hat{\pi} = \underset{\pi \in \Pi}{\operatorname{argmin}} \, \mathbb{E}_{s \sim \mathcal{D}_{\pi^*}} [\mathcal{L}(\pi(s), \pi^*(s))]. \tag{4.11}$$

Data Aggregation. DAGGER is an iterative algorithm that improves the learner by executing a mixed behavior policy combined with the learner and the expert. At each iteration, the collected data will be aggregated into an accumulated dataset and the learner will be trained by all the data collected from previous iterations.

4.3 Methodology

In this section, we present our methodology for learning chordal extensions. In this work, we focus on how to learn elimination rules for graph elimination via imitation learning. More precisely, we propose an imitation learning scheme that mimics the elimination orderings provided by the ordering heuristics we choose.

4.3.1 MDP formulation

We begin by formulating graph elimination as a Markov decision process. First, the state space $\mathcal{S}$ is the set of simple undirected graphs. Then, for a given graph $G = (V, E)$, the corresponding set of possible actions is identified by the nodes of the graph. Transitions are

deterministic: if an action $a = v$ is performed in state G, i.e., if node v is eliminated from graph G, then the new state is uniquely defined as the graph obtained from the elimination of node v. Note that the number of nodes decreases by one at each step. Hence, even though the initial graph may be of arbitrary size, trajectories are always finite.

Thus, a policy π maps a graph to a probability distribution over its set of nodes V. Therefore, if $V = \{1, ..., n\}$, then $\pi(G)$ is a n-dimensional non-negative vector, whose i-th coordinate denotes the probability that node i be eliminated.

Finally, one may select a cost function according to the problem at hand, for example, the number of additional edges, i.e., fill-in. In that case, finding a policy that minimizes the expected total cost reduces to finding a policy that yields minimum chordal extensions. Rather than trying to minimize fill-in, which is an NP-hard problem for which efficient heuristics already exists, we adopt a more generic imitation learning scheme as discussed below.

4.3.2 Learning mechanism

Although the goal of imitation learning is to find a learner policy that best matches the expert, any parameterized stochastic policy will inevitably have chances to make occasional mistakes by choosing an action different from the expert. In the supervised imitation learning approach, where the learner is only trained under the distribution of states induced by the expert, the learner may not be able to correct its behavior from deviations induced by its bad choice of action.

Moreover, any possible state is observable in our problem setting and the expert is always accessible for querying any state. Therefore, we choose the learner as the behavior policy in order to alleviate the deviation problem. As a result, the goal is to find a policy $\hat{\pi}$ such that

$$\hat{\pi} = \underset{\pi \in \Pi}{\operatorname{argmin}} \, \mathbb{E}_{s \sim \mathcal{D}_\pi}[\mathcal{L}\left(\pi(s), \pi^*(s)\right)]. \tag{4.12}$$

Since, in general, the expected loss cannot be computed analytically, we estimate it by sampling trajectories of states from finite dataset using π. Given a dataset $\mathcal{G}$ of M graphs, we train the learner with our practical algorithm, namely *one-step imitation learning*, as

described in Algorithm 1.

Algorithm 1: One-step imitation learning

Input: Instance Dataset $\mathcal{G} = \{\mathbf{g}_i\}_{i=1}^{M}$

Initialize π_θ to any policy in Π;

for $i = 1$ **to** N **do**

  for each $g \in \mathcal{G}$ **do**

    Initialize s;

    for $j = 1$ **to** T **do**

      $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}\left(\pi_\theta(s), \pi^*(s)\right)$;

      $a \leftarrow \pi_\theta(s)$;

      Take action a, observe next state s;

    end

  end

end

return π_θ

The training proceeds as follows. At every epoch i (from 1 to N), each instance $G = (V, E) \in \mathcal{G}$ is used to generate one complete trajectory (with length $T = |V|$), and each state (i.e., graph) in this trajectory represents a training data point. The one-step updating is as follows. At every transition, i.e., at every elimination step, the learner π_θ is updated by taking a gradient step with respect to the loss $\mathcal{L}\left(\pi_\theta(s), \pi^*(s)\right)$, where s is the current state. An action a is then sampled from the updated learner π_θ, which yields the next state s'. Note that, in this setting, the loss is computed by comparing the two distributions $\pi_\theta(s)$ and $\pi^*(s)$ directly. We do so because 1) we know analytically both the learner and the expert, and 2) it allows to exploit information from the entire distributions rather than sampling and comparing individual actions.

Expert. We first consider the minimum degree heuristic [57, 72] as the expert policy. At each step, the minimum degree selects a node of minimum degree to be eliminated. Ties are broken arbitrarily, i.e., if several nodes have minimum degree, then one is selected uniformly at random among them. Let us note that today's implementations include several additional features, such as smarter tie breaking or the simultaneous elimination of multiple nodes.

In all that follows, we denote π_{MD} the minimum degree policy, i.e., for a given graph

$G = (V, E)$, we have

$$\pi_{MD}(G)[v] = \begin{cases} \frac{1}{k} & \text{if } v \text{ is of minimum degree} \\ 0 & \text{otherwise} \end{cases},$$

where k is the number of nodes that have minimum degree.

When the expert is the minimum degree policy, we can compute the exact $\pi^*(s) = \pi_{MD}(s)$ given a state s of a graph as shown before. Other heuristics can be used should one want to.

Learner parameterization. Given that states are represented as graphs, with arbitrary size and topology, we propose to use *graph neural networks* (GNNs) [73, 74] to parameterize the learner. Indeed, GNNs are an expressive class of models to process graph-structured data, and have been applied to a variety of representation learning tasks on graphs [27, 75–78]. In this work, we describe GNN as a function f that takes the adjacency matrix $\mathbf{A}$ of a given graph $G = (V, E)$ and the feature matrix $\mathbf{X}$ as input. Then, f maps the input $(\mathbf{A}, \mathbf{X})$ to a probability distribution $\mathbf{Y}$ over V. In practice, a GNN model is commonly built by connecting multiple layers in a chain, with layer k defined by a function g^k. For example, given g^0, g^1, g^2 as layer-wise functions, a 3-layer GNN model can be formulated by

$$f(\mathbf{A}, \mathbf{X}) = g^2(\mathbf{A}, g^1(\mathbf{A}, g^0(\mathbf{A}, \mathbf{X}))). \tag{4.13}$$

GNN models embed both the features of nodes and the topological structure of the graph, which makes them an appropriate class of models for our problem. Another appealing property of GNNs is that they are size-and-order invariant to input data, i.e., they can process graphs of arbitrary size, and the ordering of the input elements is irrelevant.

Specifically, our GNN architecture applies the following layer-wise function g, in order to compute the embedding in the $(l+1)$-th layer from the previous layer, i.e.,

$$\mathbf{H}^{l+1} = g^l(\mathbf{A}, \mathbf{H}^l) \tag{4.14}$$

$$= \sigma^l\left(\mathbf{A}\mathbf{H}^l\mathbf{W}^l + \mathbf{I}_{n\times 1}\mathbf{B}^l\right), \tag{4.15}$$

where $\mathbf{A}\in\mathbb{R}^{n\times n}$ is the adjacency matrix of the graph, $\mathbf{I}_{n\times 1}$ is the matrix of ones with size of $n \times 1$, d^l and d^{l+1} are the dimension of the features in layer l and $(l+1)$, $\mathbf{H}^{l+1}\in\mathbb{R}^{n\times d^{l+1}}$ and $\mathbf{H}^l\in\mathbb{R}^{n\times d^l}$ are the embeddings of layer $(l+1)$ and l, $\mathbf{W}^l\in\mathbb{R}^{d^l\times d^{l+1}}$ and $\mathbf{B}^l\in\mathbb{R}^{1\times d^{l+1}}$ are the parameters in layer l, and $\sigma(\cdot)$ specifies the activation function. For $\sigma(\cdot)$, we apply Softmax in the output layer and ReLU in the rest.

Given the input $\mathbf{X} \in \mathbb{R}^n$, the Softmax and ReLU functions are defined as

$$\text{Softmax}(\mathbf{X})_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}, \tag{4.16}$$

$$\text{ReLU}(\mathbf{X})_i = \max(0, x_i), \tag{4.17}$$

where $i = 1, ..., n$.

Loss function. To measure the distance between two distributions given by the expert and the learner, we compute the Kullback-Liebler (KL) divergence [79] between $\pi^*(s)$ and $\pi_\theta(s)$, respectively,

$$\mathcal{L}_{KL}(\pi^*(s) \parallel \pi_\theta(s)) = \sum_{a \in \mathcal{A}} \pi^*(s, a) \log \frac{\pi^*(s, a)}{\pi_\theta(s, a)}. \tag{4.18}$$

If $\pi^*(s, a)$ is zero, the corresponding term in the summation is taken to be zero. Note that, for the class of GNNs that we consider, $\pi_\theta(s, a)$ is always positive by definition of Softmax. Therefore, the KL loss is always finite, though it may be arbitrarily large.

4.4 Numerical experiments

In this section, we report the details of our computational investigation. More precisely, Section 4.4.1 specifies the data generation and collection. In Section 4.4.2, we discuss the experimental setting and, finally, Section 4.4.3 reports the computational results.

4.4.1 Data collection

We evaluate our approach on four different datasets, which comprise graphs that vary in size and structural characteristics.

Erdos-Renyi graphs

We first build two datasets of Erdos-Renyi graphs, a simple and well-known class of random graphs. We use the notation $G(n, p)$ to denote a (random) Erdos-Renyi graph with n nodes, and such that edges are selected with probability $p \in [0, 1]$ independently of each other. Note that, for given n and p, $G(n, p)$ is a random variable whose realizations are graphs of size n. While n controls the size of the graph, p controls its sparsity.

We form two datasets of Erdos-Renyi graphs: one of smaller graphs, denoted ER_S, and the other one of larger graphs, denoted ER_L.

Each graph in ER_S, is sampled from $G(n, p)$, where n is drawn uniformly between 100 and 300, and p is sampled between 0.1 and 0.3 with uniform probability. This is done to introduce some variability in size and density in the dataset. Overall, ER_S contains 600 graphs. We follow the same methodology for ER_L, except that n is drawn uniformly between 300 and 500. Overall, ER_L contains 200 graphs.

SuiteSparse matrix collection

The SuiteSparse matrix collection[1] [80] is a dataset of (sparse) matrices collected from a number of real-life applications, and is routinely used as benchmark for numerical linear algebra software. Given a matrix M, we construct a non-oriented graph whose adjacency matrix corresponds exactly to the sparsity structure of M. We only consider square matrices, and any non-symmetric matrix is transferred into symmetric by adding its transpose to it.

First, we select square matrices of size between 50 and 500. This yields a dataset of 278 graphs, which we denote by SS_S. Similarly, we select square matrices of size between 1000 and 2000, and obtain a second dataset, denoted by SS_L, which contains 295 graphs.

4.4.2 Experimental settings

Our experiments were conducted on a dual Intel Xeon Gold 6126@2.60GHz, 768BG RAM machine running Linux and equipped with Nvidia Tesla V100 GPUs. Our code[2] is written in Python 3.6, and we use Pytorch 0.4 for modeling and training GNNs.

Datasets. We split the ER_S dataset into $\{training,\ validation,\ test\}$, each containing 200 graphs. Our GNN policy is trained and validated only with the *training* and *validation* set of ER_S, respectively. Then, we test the generalization performance of the trained model with the *test* set of ER_S, ER_L, SS_S and SS_L.

GNN setting. We apply the GNN architecture described in Section 4.3.2 with 2 layers. As initially the vertices of the graphs have no attribute, we initialize the feature of each vertex with the same value. Specifically, we take $h_v^0 = 1,\ \forall v \in V$. As a result, the encoding of each vertex only depends on the topological structure of its neighborhoods. The dimension

[1] SuiteSparse matrix collection was formerly known as the University of Florida sparse matrix collection.
[2] `https://github.com/ds4dm/GraphRL`

of features in all layers is the same. For each layer, the weights are initialized from Xavier normal distribution [81] and we initialize the bias with zero.

Performance metrics. In our imitation learning scheme, the goal is to minimize the divergence between the distributions produced by the learner and the heuristic expert. Since the KL divergence is capable of identifying the distance between two probability distributions, we use it as a measure of the dissimilarity of policies, in order to assess the performance of our learning scheme. Therefore, for a finite dataset $\mathcal{G}$ and expert policy π^*, the first metric computes the average KL loss, given by

$$\hat{\mathcal{L}}_{KL} = \frac{1}{\sum_{g\in\mathcal{G}} n_g} \sum_{g\in\mathcal{G}} \sum_{i=1}^{n_g} \mathcal{L}_{KL}(\pi^*(s_i) \parallel \pi_\theta(s_i)), \tag{4.19}$$

where n_g is the size of each graph $g \in \mathcal{G}$ and $\mathcal{L}_{KL}(\cdot)$ specifies the KL divergence between $\pi^*(s_i)$ and $\pi_\theta(s_i)$ in state s_i of g.

To measure the fill-in of a policy, the second metric computes the average number of fill-in per graph. For each graph $g \in \mathcal{G}$, we denote the total number of fill-in by c^g_{fillin}. Then, the average fill-in per graph is given by

$$\hat{C}_{fillin} = \frac{1}{|\mathcal{G}|} \sum_{g\in\mathcal{G}} c^g_{fillin}. \tag{4.20}$$

Training and validation settings. We train our GNN policy with Algorithm 1. At each epoch, we randomly shift the training set and sample single-graph mini batches. For learning rate tuning, we experiment different learning rates from 10^{-5} to 10^{-3}. The validation result is shown by plotting the average KL loss and the average fill-in per graph in Figure 4.1. Observing that 10^{-4} yields fast and smooth convergence, we train the model with the learning rate of 10^{-4} for 20 epochs. Moreover, we also observe a plateau effect for larger step size in Figure 4.1, notably, sudden decrease with larger step size. This effect will be discussed in Section 4.5.

Test. We first test the performance of GNNs trained on the training set of ER_S with four test sets as specified before. To evaluate the performance of GNN models at different stages of training, we first save the trained model at the end of each epoch. Then, we test the performance of each saved model on four test sets. Furthermore, we also train GNNs with SS_S and compare its performance with that of the policy trained from ER_S on all the test sets, in order to analyze the impact of the choice on training set.

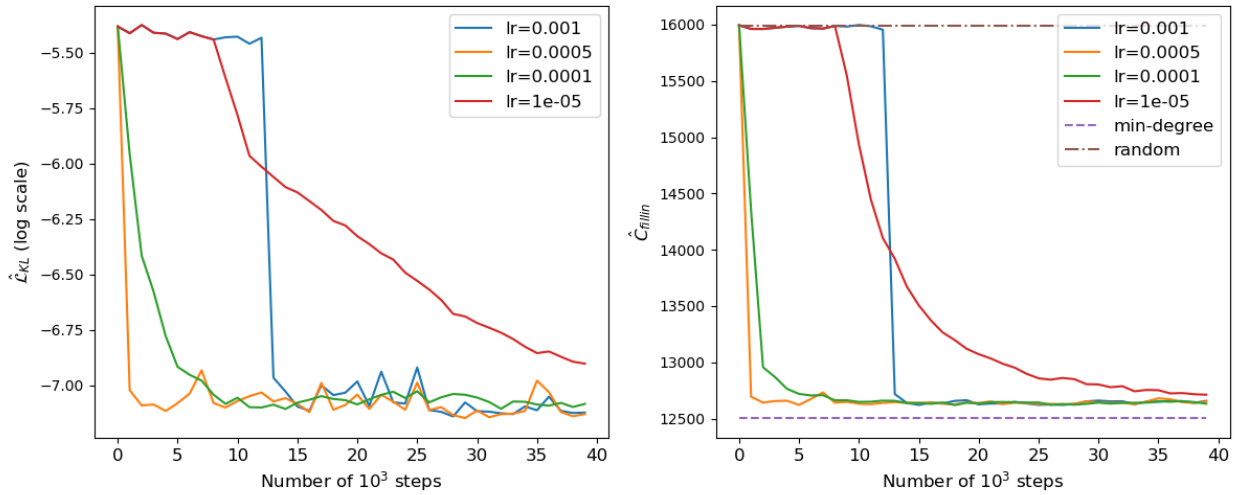

Figure 4.1 Validation results of imitation learning. We plot the average KL loss in *log* scale (left) and the average fill-in per graph (right) on the validation set of ER_S. For fill-in, we compare GNN with minimum degree and random policy.

Imitation of another heuristic. To analyze the performance of our framework on learning other heuristics than minimum degree, we apply it to the so-called minimum fill-in heuristic [82]. Instead of eliminating the node of minimum degree, minimum fill-in chooses the node such that the number of added edges at each step is minimized. For this experiment, we apply a GNN architecture with three layers. We also implement the minimum fill-in heuristic as the expert policy. The other settings are the same as the previous experiment of learning minimum degree.

4.4.3 Results

Imitation of minimum degree heuristic

In this section, we compare the predictive performance of GNN with the two metrics introduced in the previous section. Let π_{ER_S} denote the policy obtained by training over the training set of ER_S. The results of π_{ER_S} on the training set and four test sets are shown in Figure 4.2. Specifically, we plot the curves of two metrics over the entire training period (20 epochs), in order to compare the performance of GNN models at different stages of training.

From the results of the training set (shown in the first row of Figure 4.2), we observe that the loss significantly decreases and stabilizes after about 15 epochs of training. Moreover, for fill-in, the GNN also matches the minimum degree heuristic.

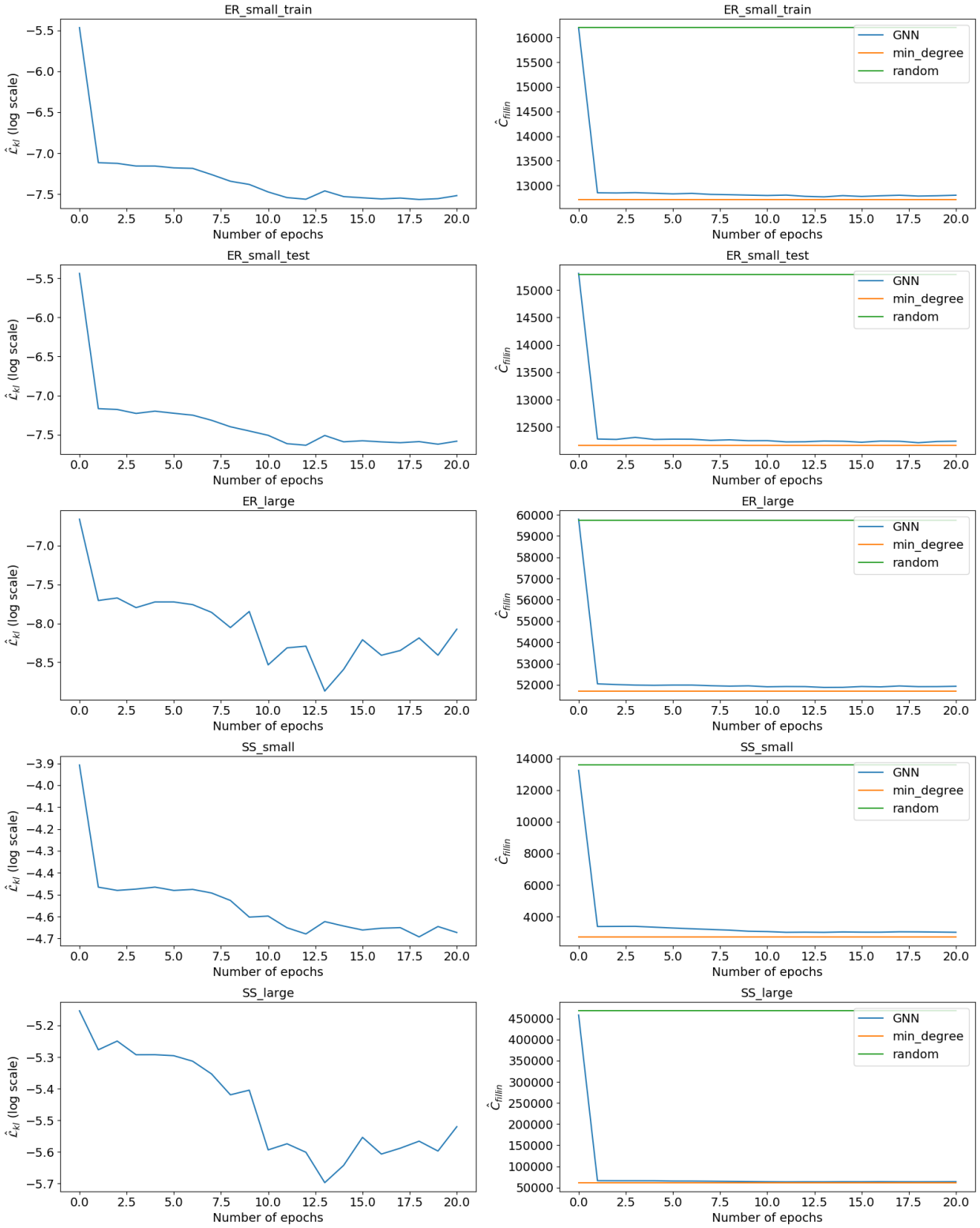

Figure 4.2 Evaluation of π_{ER_S} over different datasets. We plot the average KL loss in *log* scale (left) and the average fill-in per graph (right) on the training set and four test sets. For fill-in, we compare our GNN with minimum degree and random policy.

Comparing the results of different test sets, we observe that our GNN generalizes well, both to larger graphs and to different distributions. First of all, loss curves on all test sets show same decreasing tendency over the entire training period, although the magnitude of values can be different across datasets. In terms of fill-in, we have similar and consistent results. Moreover, by comparing the curves in each row, we also observe a strong correlation between KL loss (i.e., how good we replicate the minimum degree expert), and the actual fill-in (which is only observed, we never learn anything from it).

It is worth noting that, although the initial graphs can be i.i.d., the other states in the trajectory always depend on previous states and actions, which indicates the induced distribution of states depends on the behavior policy itself.

Since we have chosen the learner as the behavior policy in our approach, the loss is measured under the distribution of states induced by GNN policies. As the GNN model changes during training, this metric is actually measured under different distributions. As a result, the decrease of loss over the training period (shown on the left side of Figure 4.2) only shows a tendency that the GNN replicates the minimum degree expert better on an evolutionary distribution induced by itself. The predictive performance of the GNN still needs to be validated by the actual fill-in, which is precisely done on the right side of Figure 4.2.

Next, we evaluate the impact of the choice of training set on the generalization performance of learned policies. Let π_{SS_S} denote the policy obtained by training over SS_S dataset, and π_U denote a uniform policy, i.e., one that selects each node with uniform probability. In Table 4.1, we report the average KL loss of each policy, over each of the four test sets ER_S, ER_L, SS_S and SS_L. Recall that, since the KL loss is computed along trajectories, its value depends on the behavior policy. Therefore, to make the comparison of policies valid, the values reported in each column of Table 4.1 are computed using the *same* set of trajectories, generated by the expert policy π_{MD}. Policies that are obtained via imitation learning should have lower KL loss than π_U and, by definition, the KL loss of the expert policy π_{MD} is zero. Indeed, both π_{ER_S} and π_{SS_S} display better KL loss than π_U, with π_{SS_S} performing better than π_{ER_S} on all four test sets.

To validate these findings, in Table 4.2, we report the average fill-in of each policy over the same four test sets; the fill-in of learned policies should match that of the expert π_{MD}. The results of Table 4.2 corroborate those of Table 4.1: the fill-in of π_{SS_S} deviates from that of π_{MD} by no more than 1%, π_{ER_S} by up to 2.5% on the SS_L test set, while π_U deviates by as much as 650% on SS_L. Importantly, this last example shows that relatively small variations in KL loss may result in very large fluctuations of fill-in. One likely explanation is that, in the earlier steps, a wrong decision may result in substantial fill-in, which then propagates as

more nodes are eliminated. Furthermore, results for the ER_L test set show that, even though the KL loss of π_{SS_S} is about one order of magnitude smaller than that of π_{ER_S}, both policies achieve very similar fill-in with a relative difference of only 0.3%. Overall, we conclude that training on ER graphs generalizes well to both graphs of larger size and from a different distribution, displaying only a marginal loss of performance. This last point is especially important in contexts where realistic data is scarce, since ER graphs can be generated easily and in a controlled way.

Imitation of minimum fill-in heuristic

In this section, we analyze the results of our approach on learning the minimum fill-in heuristic. We first report the learning curve on the training set of ER_S in Figure 4.3. We can observe that the average KL loss decreases notably after a few epochs of training and the average fill-in of GNN policy converges close to that of minimum fill-in policy.

Moreover, we analyze the generalization performance of trained GNN on different test sets. Since running the minimum fill-in heuristic on SS_L takes too much time, here we only report results over ER_S, ER_L and SS_S. Let π_{GNN} denote the GNN policy obtained by training over the training set of ER_S, and π_{MF}, π_U denote the minimum fill-in and uniform policy, respectively. In Table 4.3, we report the average KL loss of π_{GNN} and two baseline policies. Similar to Table 4.1, the values of KL loss reported in each column of Table 4.3 are computed along the same set of trajectories generated by the expert policy π_{MF}. The results show that π_{GNN} has lower KL loss than π_U on all test sets.

To validate the fill-in performance of the trained GNN policy, we report the average fill-in of π_{GNN} and two baselines in Table 4.4. For each test set, the average fill-in of each policy is computed along the trajectories generated by the evaluated policy itself. Results on all test sets show that the fill-in of π_{GNN} deviates from that of π_{MF} by no more than 5%, while π_U

Table 4.1 Average KL loss of different policies over various test sets. For each test set, the average KL loss of policies is computed using the *same* trajectories, generated by the policy π_{MD}.

	Test set			
Policy	ER_S	ER_L	SS_S	SS_L
π_U	5.67×10^{-3}	1.85×10^{-3}	2.07×10^{-2}	7.18×10^{-3}
π_{ER_S}	5.46×10^{-4}	3.33×10^{-4}	8.97×10^{-3}	5.65×10^{-3}
π_{SS_S}	2.24×10^{-5}	9.36×10^{-6}	4.79×10^{-4}	2.89×10^{-3}

Table 4.2 Average fill-in of different policies over various test sets. For each test set, the average fill-in of policies is computed using the trajectories generated by the evaluated policy itself.

	Test set			
Policy	ER_S	ER_L	SS_S	SS_L
π_{MD}	$12,165$	$51,711$	$2,719$	$61,405$
π_U	$15,286$	$59,753$	$13,588$	$468,306$
π_{ER_S}	$12,223$	$51,915$	$2,877$	$62,957$
π_{SS_S}	$12,189$	$51,755$	$2,809$	$62,096$

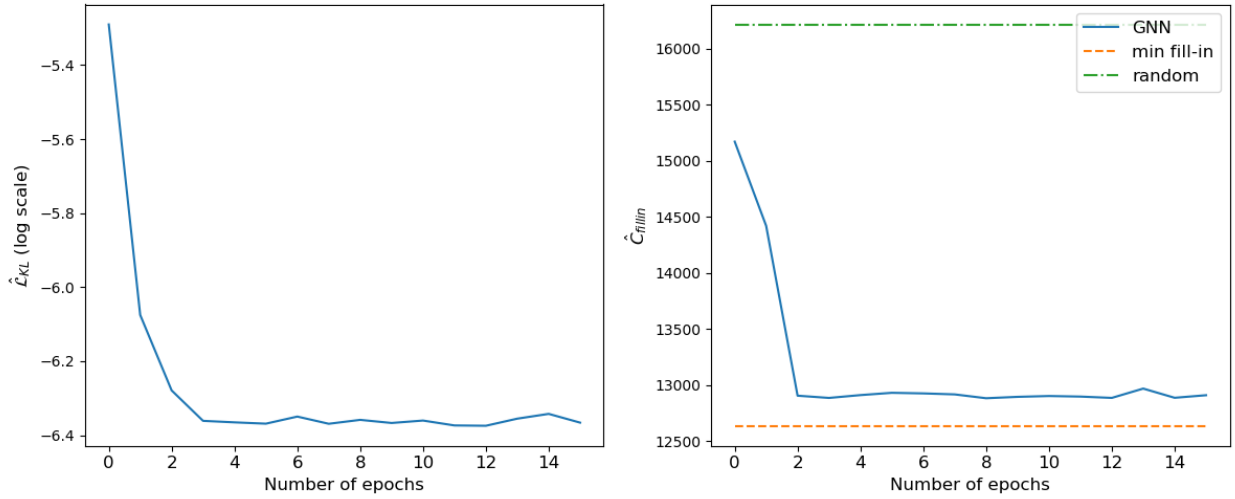

Figure 4.3 Learning curve of GNN for imitating the minimum fill-in heuristic. We plot the average KL loss in *log* scale (left) and the average fill-in per graph (right) on the training set of ER_S. For fill-in performance, we compare GNN with minimum fill-in and random policy.

Table 4.3 Average KL loss of policies over test sets. For each test set, the average KL loss of policies is computed using the *same* trajectories, generated by the policy π_{MF}.

	Test set		
Policy	ER_S	ER_L	SS_S
π_U	7.77×10^{-3}	2.67×10^{-3}	2.51×10^{-2}
π_{GNN}	5.40×10^{-3}	1.64×10^{-3}	1.85×10^{-2}

deviates by as much as 443% on SS_S. Therefore, we conclude that the GNN policy trained on small Erdos-Renyi graphs is able to replicate the minimum fill-in heuristic on the same class of graphs, and generalizes well to both lager sized graphs and graphs from a different distribution.

4.5 Further discussion

We now seek to further explain the sharp drops in loss that were observed during training, e.g., in Figure 4.1, and the stark correlation between imitation loss and fill-in.

To do so, we consider the following GNN with two layers:

$$x_i^0 = 1 \qquad \forall i \in V, \tag{4.21}$$

$$h_i^1 = \sum_{j \in \mathcal{N}(i)} w_1 x_j^0 \qquad \forall i \in V, \tag{4.22}$$

$$x_i^1 = \mathrm{ReLU}(1 + h_i^1) \qquad \forall i \in V, \tag{4.23}$$

$$h_i^2 = \sum_{j \in \mathcal{N}(i)} w_2 x_j^1 \qquad \forall i \in V, \tag{4.24}$$

$$x^2 = \mathrm{Softmax}(h^2), \tag{4.25}$$

where $w_1, w_2 \in \mathbb{R}$ are the only two scalar parameters of the GNN, and x^0, h^1, x^1, h^2, x^2 are vectors of size $|V|$. The input vector is x^0 with all coordinates equal to one and recall that, by definition of Softmax, the coordinates of the output vector x^2 are all non-negative and sum to one. Although this corresponds to a slightly simpler model than that of Section 4.4.3, it gives several insights into the behavior during training. Finally, for given w_1, w_2, we denote the corresponding policy by π_w.

Table 4.4 Average fill-in of policies over test sets. For each test set, the average fill-in of policies is computed using the trajectories generated by the evaluated policy itself.

	Test set		
Policy	ER_S	ER_L	SS_S
π_{MF}	12,109	51,564	2,503
π_U	15,286	59,753	13,588
π_{GNN}	12,136	52,245	2,625

4.5.1 Policy interpretation

We begin by plotting the landscape of the expected average KL loss, evaluated on the training set. This landscape is represented in Figure 4.4.

First, for every node i, since $x_i^0 = 1$, we have $h_i^1 = w_1\delta(i)$. It follows that, by setting $w_1 = 0$, we get $x_i^1 = \text{ReLU}(1+0) = 1$, and then $h_i^2 = w_2 \times \delta(i)$. Therefore, as w_2 approaches $-\infty$, x^2 becomes arbitrarily close to a minimum degree distribution, and π_w becomes arbitrarily close to π_{MD}. Indeed, the average loss is minimized when $w_1 = 0$ and w_2 goes to $-\infty$.

Second, we observe that in the $w_1 \leq -1$ region, the average loss is flat. This region actually corresponds to the ReLU of the first layer being inactive. Indeed, we have $h_i^1 = w_1\delta(i)$, therefore, when $w_1 \leq -1$, we automatically get $1 + h_i^1 \leq 0$, which yields $x_i^1 = 0$. Consequently, the output of the GNN is a uniform distribution on the nodes of the graph, i.e., we obtain $x_i^2 = \frac{1}{n}$ for each node $i \in \{1, ..., n\}$.

Third, observe that the landscape of the loss function displays fairly flat regions, which tend to be separated by sharp drops in the objective, e.g., around the $w_1 = 0$ region. This landscape most likely explains the shapes of the training curves in Figure 4.1, which displayed flat progression followed by sharp drops in the loss. Whether such behavior would carry out in larger dimensions remains an open question.

This manual inspection of a GNN policy becomes intractable when using, e.g., numerous layers, or multiple features within each layer. Nevertheless, recent works such as [83] have developed auxiliary tools for interpreting GNN models, namely by identifying a small sub-graph and subset of node features that play an essential role in the GNN prediction. Further qualitative insight can be gained by choosing a set of sample graphs, and directly visualizing the policy's decision for those graphs. Note that doing so does not require any assumption about the policy's parametrization. This approach is illustrated in Figure 4.5, where we display the policy $\pi_{-1,1}$ on four sample graphs. For the top-left graph, which is a tree, the policy assigns an identical probability to all four leaves, and a lower probability to the other three nodes. In Figure 4.6, we also visualize the chordal extensions of an Erdos-Renyi graph built by GNN and minimum degree policy. In this example, both policies make identical decisions at the first three steps, while the decisions become different from step 4. However, the graphs are complete at this point, so the different orderings afterwards have no impact on the final chordal extensions.

Finally, in addition to the elimination rule itself, one may study the characteristics of chordal extensions that are produced by it, e.g., by comparing some features of interest such as number of additional edges, number of cliques and their sizes, etc. Such an *a posteriori* analysis is

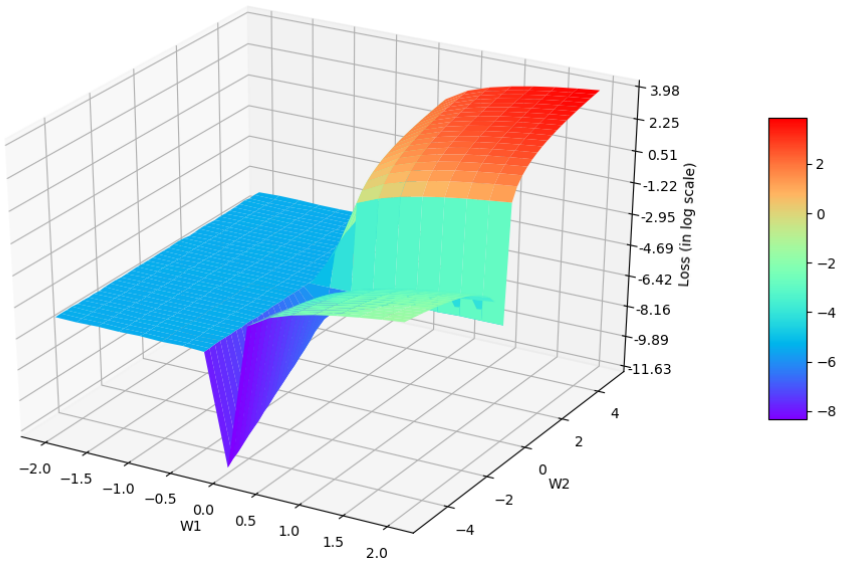

Figure 4.4 Landscape of the expected average KL loss (in log scale). For each (w_1, w_2), we plot the expected average KL loss, estimated over the training set.

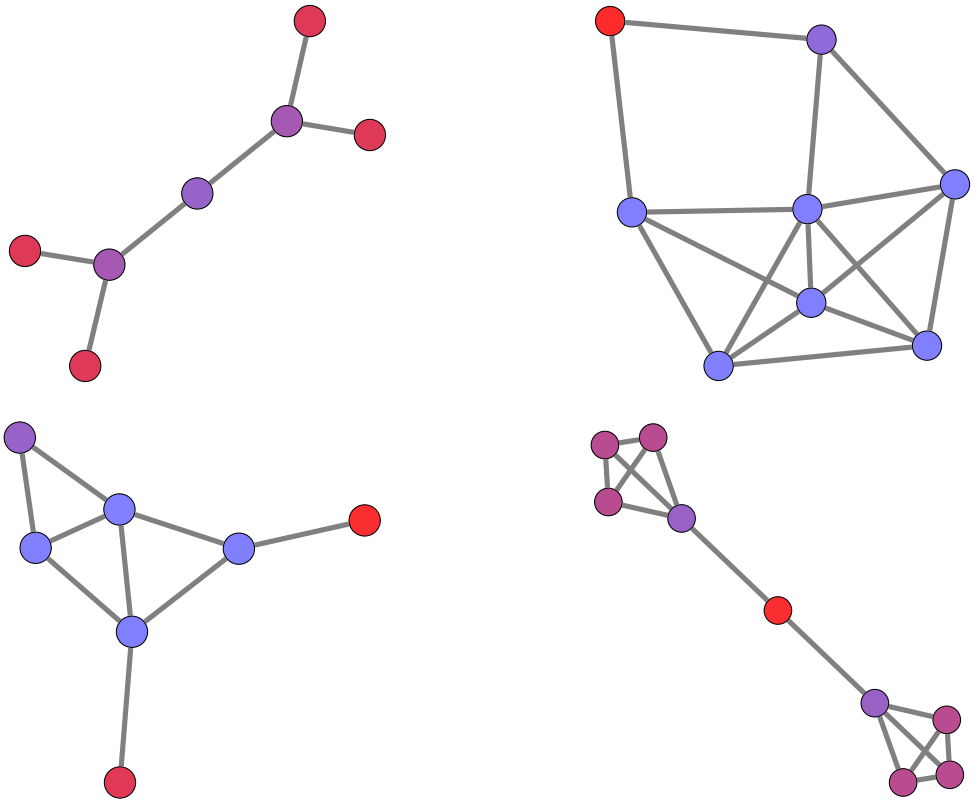

Figure 4.5 Visualization of π_w on four sample graphs, with $w = (-1, 1)$. Nodes that are assigned a higher (resp. lower) probability of elimination are indicated in red (resp. blue).

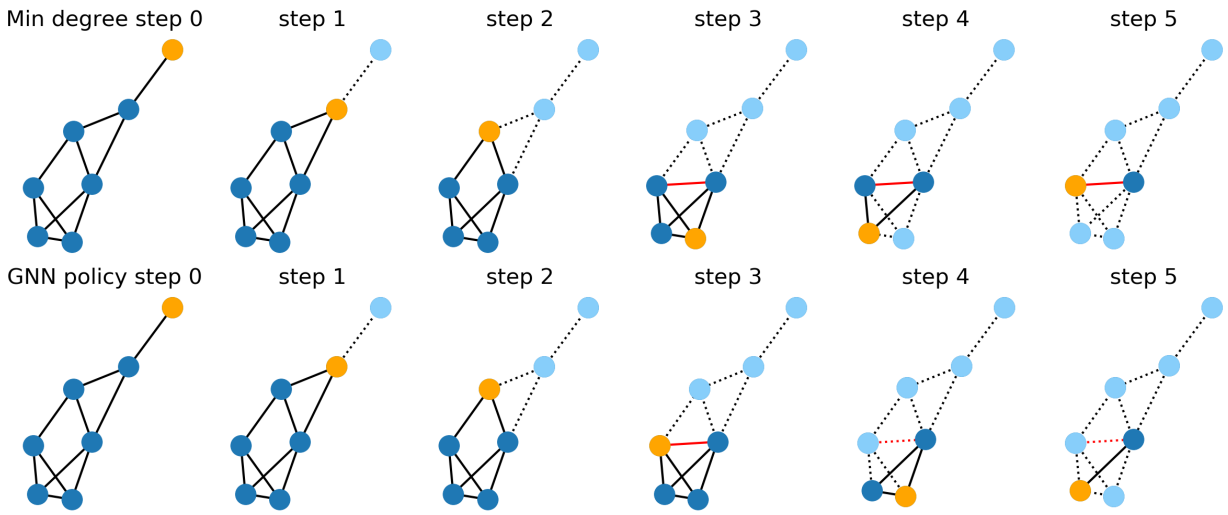

Figure 4.6 Visualization of building chordal extensions of an Erdos-Renyi graph by minimum degree heuristic and GNN. The first row plots the solution built by the min degree policy, whereas the second row plots a solution built by GNN. At each step, the selected node is colored in orange. New added edges are in red. The eliminated nodes are in light blue and the removed edges are dashed.

most relevant in the context of RL-based approaches, where the learned policy – and the corresponding chordal extensions – may differ substantially from known experts. This can be performed with GNN models, or classical feature-based ML algorithms such as Random Forests or Support Vector Machines. Note that, in the latter case, the set of features needs to be engineered manually.

4.5.2 Relation to fill-in

We then plot the landscape of the expected total fill-in in Figure 4.7, also evaluated on the training set. While this gives us an insight into how fill-in correlates with the KL loss, let us formally restate that fill-in is never used during the training process. In particular, no gradient information is ever inferred from fill-in.

First, unsurprisingly, similar to Figure 4.4, here we observe a flat landscape in the $w_1 \leq -1$ region. Recall indeed that setting $w_1 \leq -1$ means the GNN's output reduces to a uniform policy. Second, the region $w_1 \geq 0, w_2 \geq 0$ displays high fill-in. This is not surprising either since this region essentially yields policies that select nodes with high degree, which is naturally detrimental to fill-in.

A third and more remarkable observation is the flat valley in the region $w_1 \geq 0, w_2 \leq 0$. While

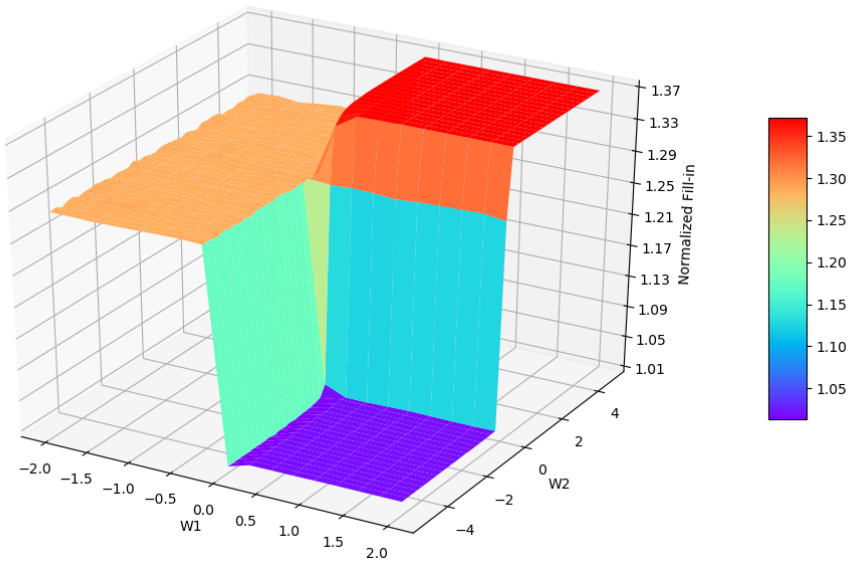

Figure 4.7 Landscape of the normalized total fill-in. For each (w_1, w_2), we plot the average total fill-in of the corresponding GNN policy, divided by the expected total fill-in of the minimum degree heuristic. Both expectations are estimated over the training set.

we know that the GNN policy converges to minimum degree when $w_1 = 0$ and w_2 takes large negative values, the plots in Figure 4.7 show that, when it comes to fill-in, the magnitude of w_2 does not matter as much.

Fourth and last, among the set of policies $\Pi_{GNN} = \{\pi_w | w \in \mathbb{R}^2\}$, the lowest fill-in is achieved for $w_1 \geq 0, w_2 \leq 0$. Recall that, as mentioned in Section 4.5.1, the minimum degree policy π_{MD} corresponds to π_{0,w_2} as w_2 approaches $-\infty$. Thus, according to Figure 4.7, π_{MD} appears to be a minimizer of the fill-in, among the set of policies Π_{GNN}. Consequently, for the considered set of graphs and class of models, we observe that no policy π_w can outperform π_{MD} in terms of fill-in. Let us emphasize that this holds independently of the learning procedure. Thus, models with higher representation power, e.g., GNNs with additional layers, would be required to achieve lower fill-in.

4.6 Conclusion

In this work, we have considered chordal extensions and graph elimination as major factors for devising sparsity-exploiting techniques for optimization algorithms. We have argued that, although effective heuristics to perform graph elimination (an NP-complete task) exist, there is no definitive understanding of the effect of the obtained chordal extension on the optimization algorithm using the final graph representation.

For this reason, we have followed the current research trend of looking at Combinatorial Optimization tasks by using a Machine Learning lens and we have devised a framework for learning elimination rules yielding high-quality chordal extensions. As a first building block of the learning framework, we have proposed an imitation learning scheme, which we have illustrated on two classical elimination rules: Minimum Degree and Minimum Fill-in.

The results have shown that our imitation learning approach is effective in learning both experts, using simple GNN models with only a handful of parameters. We have observed that learning elimination rules displays remarkable generalization performance: the learned policies successfully extend to graphs of larger size, and to graphs from a different distribution. Importantly, training on a synthetic dataset of small Erdos-Renyi graphs results in a marginal loss of performance, a desirable behavior since it allows to speed-up the learning process by training on smaller problems. Finally, we have discussed various ways to interpret the learned elimination rules, whether by inspecting the learned policy itself, or by studying the features of the obtained chordal extensions, the latter being more relevant to RL-based approaches.

We identify two main research avenues for subsequent developments. On one hand, while GNNs are a good model prior for combinatorial problems over graphs, enlarging their representation power, for instance to represent hypernodes or to model multiple eliminations, will likely be key to handling practical tasks. On the other hand, the next logical step will be to learn elimination rules that explicitly address the performance of practical optimization algorithms, in conjunction with reinforcement learning-based approaches. We expect that a combination of learned policies and a feature-based analysis of the resulting chordal extensions can lead to an easier and more direct customization of elimination orderings to sets of similar graphs. In that regard, our future work will investigate chordal decomposition specially tailored to SDP optimization problems.

CHAPTER 5 ARTICLE 2: REVISITING LOCAL BRANCHING WITH A MACHINE LEARNING LENS

Authors: Matteo Fischetti, Defeng Liu, Andrea Lodi
Submitted to *Mathematical Programming Computation*. Submission date: 28 July, 2022.

Abstract Finding high-quality solutions to mixed-integer linear programming problems (MILPs) is of great importance for many practical applications. In this respect, the refinement heuristic *local branching* (LB) has been proposed to produce improving solutions and has been highly influential for the development of local search methods in MILP. The algorithm iteratively explores a sequence of solution neighborhoods defined by the so-called *local branching constraint*, namely, a linear inequality limiting the distance from a reference solution. For a LB algorithm, the choice of the neighborhood size is critical to performance. In this work, we study the relation between the size of the search neighborhood and the behavior of the underlying LB algorithm, and we devise a learning based framework for predicting the best size for the specific instance to be solved. Furthermore, we have also investigated the relation between the time limit for exploring the LB neighborhood and the actual performance of LB scheme, and devised a strategy for adapting the time limit. We computationally show that the neighborhood size and time limit can indeed be learned, leading to improved performance and that the overall algorithm generalizes well both with respect to the instance size and, remarkably, across instances.

5.1 Introduction

Mixed-integer linear programming (MILP) is a main paradigm for modeling complex combinatorial problems. The exact solution of a MILP model is generally attempted by a branch-and-bound (or branch-and-cut) [1] framework. Although state-of-the-art MILP solvers experienced a dramatic performance improvement over the past decades, due to the NP-hardness nature of the problem, the computation load of finding a provable optimal solution for the resulting models can be heavy. In many practical cases, feasible solutions are often required within a very restricted time frame. Hence, one is interested in finding solutions of good quality at the early stage of the computation. In fact, it is also appealing to discover early incumbent solutions in the exact enumerate scheme, which improves the primal bound and reduces the size of the branch-and-bound tree by pruning more nodes [84].

In this respect, the concept of heuristic is well rooted as a principle underlying the search of high-

quality solutions. In the literature, a variety of heuristic methods have proven to be remarkably effective, e.g., *local branching* [85], *feasibility pump* [86], *RINS* [87], *RENS* [88], *proximity search* [89], *large neighborhood search* [90], etc. More details of these developments are reviewed in [4, 90, 91]. In this paper, we focus on local branching, a *refinement heuristic* that iteratively produces improved solutions by exploring suitably predefined solution neighborhoods.

Local branching (LB) was one of the first methods using a generic MILP solver as a black-box tool inside a heuristic framework. Given an initial feasible solution, the method first defines a solution neighborhood through the so-called *local branching constraint*, then explores the resulting subproblem by calling a black-box MILP solver. For a LB algorithm, the choice of neighborhood size is crucial to performance. In the original LB algorithm [85], the size of neighborhood is mostly initialized by a small value, then adjusted in the subsequent iterations. Although these conservative settings have the advantage of yielding a series of easy-to-solve subproblems, each leading to a small progress of the objective, there is still a lot of space for improvement. As discussed in [89], a significantly better performance can be potentially achieved with an ad-hoc tuning of the size of the neighborhood. Our observation also shows that the "best" size is strongly dependent on the particular MILP instance. To illustrate this, the performance of different LB neighborhood size settings for two MILP instances are compared in Figure 5.1. In principle, it is desirable to have neighborhoods to be relatively small to allow for an efficient exploration, but still large enough to be effective for finding improved solutions. Nonetheless, it is reasonable to believe that the size of an ideal neighborhood is correlated with the characteristics of the particular problem instance.

Furthermore, it is worth noting that, in many applications, instances of the same problem are solved repeatedly. Problems of real-world applications have a rich structure. While more and more datasets are collected, patterns and regularities appear. Therefore, problem-specific and task-specific knowledge can be learned from data and applied to the corresponding optimization scenario. This motives a broader paradigm of learning to guide the neighborhood search in refinement heuristics.

In this paper, we investigate a learning framework for sizing the search neighborhood of local branching. In particular, given a MILP instance, we exploit patterns in both the structure of the problem and the information collected from the solving process to predict the size of the LB neighborhood and the time limit for exploring the neighborhood, with the aim of maximizing the performance of the underlying LB algorithm. We computationally show that the neighborhood size and the time limit can indeed be learned, leading to improved performances, and that the overall algorithm generalizes well both with respect to the instance size and, more surprisingly, across instances.

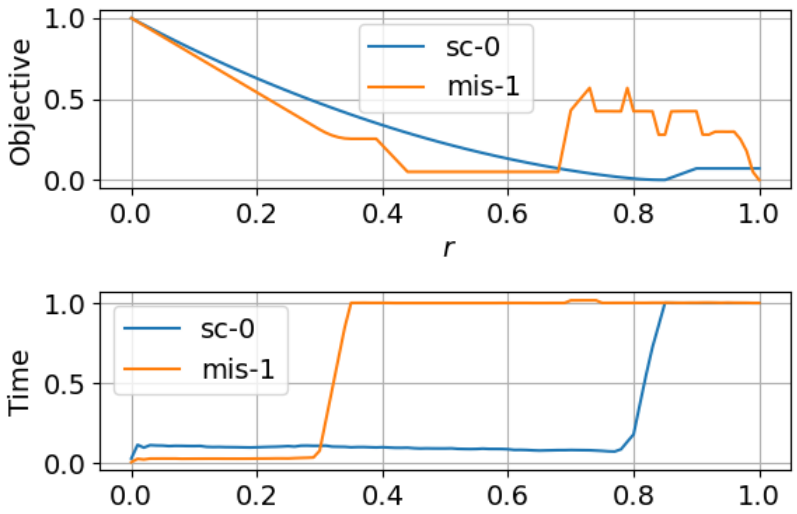

Figure 5.1 Evaluation of the size of LB neighborhood on a set covering instance (sc-0) and a maximum independent set instance (mis-1). The neighborhood size k is computed as $k = r \times N$, where N is the number of binary variables, and $r \in [0, 1]$. A time limit is imposed for each neighborhood exploration.

We note that a shorter conference version of this paper appeared in [92]. Our initial conference paper did not investigate the effect of the time limit for solving the LB neighborhood on the performance of the overall LB algorithm. This extended paper addresses this issue and provides additional analysis on applying our refined LB algorithms as a primal heuristic within the MILP sovler.

The paper is organized as follows. In Section 5.2, we give a review of the related works in the literature. In Section 5.3, we introduce the basic local branching scheme and some relevant concepts. In Section 5.4, we present our methodology for learning to search in the local branching scheme. In Section 5.5 describes the setup of our experiments and reports the results to validate our approach. In Section 5.6, we apply local branching as a primal heuristic and provide two possible implementations of how our local branching scheme interacts with the MILP solver. Section 5.7 concludes the paper and discusses future research.

5.2 Related work

Recently, the progress in machine learning (ML) has stimulated increasing research interest in learning algorithms for solving MILP problems. These works can be broadly divided into two categories: *learning decision strategies within MILP solvers*, and *learning primal heuristics*.

The first approach investigates the use of ML to learn to make decisions inside a MILP solver, which is typically built upon a general branch-and-bound framework. The learned policies

can be either cheap approximations of existing expensive methods, or more sophisticated strategies that are to be discovered. Related works include: learning to select branching variables [27,93,94], learning to select branching nodes [28], learning to select cutting planes [29], and learning to optimize the usage of primal heuristics [30, 31, 95].

The *learning primal heuristics* approach is to learn algorithms to produce primal solutions for MILPs. Previous works in this area typically use ML methods to develop *large neighborhood search* (LNS) heuristics. Within an LNS scheme, ML models are trained to predict "promising" solution neighborhoods that are expected to contain high-quality solutions. In [96], the authors trained neural networks to directly predict solution values of binary variables, and then applied the LB heuristic to explore the solution neighborhoods around the predictions. The work of [97] also uses neural networks to predict partial solutions. The subproblems defined by fixing the predicted partial solutions are solved by a MILP solver. The work of [98] proposes a LNS heuristic based on a "learn to destroy" strategy, which frees part of the current solution. The variables to be freed are selected by trained neural networks using imitation learning. Note that their methods rely on parallel computation, which makes the outcome of the framework within a non-parallel environment less clear. In [99], the authors proposed a decomposition-based LNS heuristic. They used imitation learning and reinforcement learning to decompose the set of integer variables into subsets of fixed size. Each subset defines a subproblem. The number of subsets is fixed as a hyperparameter.

Note that the learning-based LNS methods listed above directly operate on the integer variables, i.e., the predictions of ML models are at a variable-wise level, which still encounters the intrinsic combinatorial difficulty of the problem and limits their generalization performances on generic MILPs. Moreover, the learning of these heuristics is mostly based on the extraction of static features of the problem, the dynamic statistics of the heuristic behavior of the solver being barely explored. In our work, we aim at avoiding directly making predictions on variables. Instead, we propose to guide the (local) search by learning how to control the neighborhood size at an instance-wise level. To identify promising solution neighborhoods, our method exploits not only the static features of the problem, but also the dynamic features collected during the solution process as a sequential approach.

In the literature, there has also been an effort to learn algorithms for solving specific combinatorial optimization problems [20, 22, 41, 43, 100, 101]. For a detailed overview of "learn to optimize", see [16].

5.3 Preliminaries

5.3.1 Local branching

We consider a MILP problem with 0–1 variables of the form

$$\begin{aligned} \min \quad & \boldsymbol{c}^T \boldsymbol{x} && (5.1)\\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, && (5.2)\\ & x_j \in \{0,1\},\ \forall j \in \mathcal{B}, && (5.3)\\ & x_j \in \mathbb{Z}^+,\ \forall j \in \mathcal{G}, \quad x_j \geq 0,\ \forall j \in \mathcal{C}, && (5.4) \end{aligned}$$

where the index set of decision variables $\mathcal{N} := \{1, \ldots, n\}$ is partitioned into $\mathcal{B}, \mathcal{G}, \mathcal{C}$, which are the index sets of binary, general integer and continuous variables, respectively.

Note that we assume the existence of binary variables, as one of the basic building blocks of our method—namely, the local branching heuristic—is based on this assumption. However, this limitation can be relaxed and the local branching heuristic can be extended to deal with general integer variables, as proposed in [102].

Let $\bar{\boldsymbol{x}}$ be a feasible *incumbent* solution for (P), and let $\overline{\mathcal{S}} = \{j \in \mathcal{B} : \bar{x}_j = 1\}$ denote the binary support of $\bar{\boldsymbol{x}}$. For a given positive integer parameter k, we define the neighborhood $N(\bar{\boldsymbol{x}}, k)$ as the set of the feasible solutions of (P) satisfying the *local branching constraint*

$$\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) = \sum_{j \in \mathcal{B} \setminus \overline{\mathcal{S}}} x_j + \sum_{j \in \overline{\mathcal{S}}} (1 - x_j) \leq k. \tag{5.5}$$

In the relevant case in which solutions with a small binary support are considered (for example, in the famous traveling salesman problem only n or the $O(n^2)$ variables take value 1), the asymmetric form of local branching constraint is suited, namely

$$\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) = \sum_{j \in \overline{\mathcal{S}}} (1 - x_j) \leq k. \tag{5.6}$$

The local branching constraint can be used in an exact branching scheme for (P). Given the incumbent solution $\bar{\boldsymbol{x}}$, the solution space with the current branching node can be partitioned by creating two child nodes as follows:

$$\text{Left:} \quad \Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) \leq k, \qquad \text{Right:} \quad \Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) \geq k + 1.$$

5.3.2 The neighborhood size optimization problem

For a *neighborhood size* parameter $k \in \mathbb{Z}^+$, the LB algorithm obtained from choosing k can be denoted as $\mathcal{A}_k$. Given a MILP instance $\boldsymbol{p}$, with its incumbent solution $\bar{\boldsymbol{x}}$, the neighborhood size optimization problem over k for one iteration of $\mathcal{A}_k$ is defined as

$$\min \quad \mathcal{C}(\boldsymbol{p}, \bar{\boldsymbol{x}}; \mathcal{A}_k) \tag{5.7}$$

$$\text{s.t.} \quad k \in \mathbb{Z}^+, \tag{5.8}$$

where $\mathcal{C}(\boldsymbol{p}, \bar{\boldsymbol{x}}; \mathcal{A}_k)$ measures the "cost"of $\mathcal{A}_k$ on instance $(\boldsymbol{p}, \bar{\boldsymbol{x}})$ as a trade-off between execution speed and solution quality.

In practice, a run of the LB algorithm consists of a sequence of LB iterations. To maximize the performance of the LB algorithm, a series of the above optimization problems need to be solved. Since the cost function $\mathcal{C}$ is unknown, those problems cannot be solved analytically. In general, the common strategy is to evaluate some trials of k and select the most performing one with respect to the defined cost metric. This is often done by using black-box optimization methods [103]. As the evaluation of each setting involves a run of $\mathcal{A}_k$ and the best k is instance-specific, those methods are not computationally efficient enough for online use. That is why the original LB algorithm initializes k with a fixed small value and adapts it conservatively by a deterministic strategy.

Currently, learning from experiments and transferring the learned knowledge from solved instances to new instances is of increasing interest and somehow accessible. In the next section, we will introduce a new strategy for selecting the neighborhood size k by using data-driven methods.

5.4 Learning methods

Next, we present our framework for learning the neighborhood size in the LB scheme. The original LB algorithm chooses a conservative value for k as default, with the aim of generating a easy-to-solve subproblem for general MILPs. However, as discussed in Section 5.1, our observation shows that the "best"k is dependent on the particular MILP instance. Hence, in order to optimize the performance of the LB heuristic, we aim at devising new strategies to learn how to tailor the neighborhood size for a specific instance. In particular, we investigate the dependencies between the state of the problem (defined by a set of both static and dynamic features collected during the LB procedure, e.g., context of the problem, incumbent solution, solving status, computation cost, etc.) and the size of the LB neighborhood.

Our framework consists of a two-phase strategy. In the first phase, we define a regression task to learn the neighborhood size for the first LB iteration. Within our method, this size is predicted by a pretrained regression model. For the second phase, we leverage reinforcement learning (RL) [10] and train a policy to dynamically adapt the neighborhood size at the subsequent LB iterations. The exploration of each LB neighborhood is the same as in the original LB framework, and a generic black-box MILP solver is used to update the incumbent solution. The overall scheme is exact in nature although turning it into a specialized heuristic framework is trivial (and generally preferrable).

5.4.1 Scaled regression for local branching

For intermediate LB iterations, the statistics of previous iterations (e.g., value of the previous k, solving statistics, etc.) are available. One can then take advantage of this information and exploit the learning methods based on dynamic programming (e.g., reinforcement learning). Section 5.4.2 will address this case. However, for the first LB iteration, there is no historical information available as input. In this section, we will show how to define a regression task to learn the first k from the context of the problem and the incumbent solution.

Let $\mathcal{S}$ denote the set of available features of the MILPs before the first LB iteration. We aim to train a regression model $f : \mathcal{S} \to \mathbb{R}$ that maps the features of a MILP instance $\boldsymbol{s}$ to k_0^*, the label of best k_0. However, the label k_0^* is unknown, and we do not have any existing method to compute the exact k_0^*. To generate labels, we first define a metric for assessing the performance of a LB algorithm $\mathcal{A}_k$, and then use black-box optimization methods to produce approximations of k_0^* as labels.

Approximation of the best k_0

To define a cost metric for $\mathcal{A}_k$, we consider two factors. The first factor is the computational effort (e.g., CPU time) to solve the sub-MILP defined by the LB neighborhood, while the second factor is the solution quality (e.g., the objective value of the best solution). To quantify the trade-off of speed and quality, the cost metric can be defined as

$$c^{k_0} = \alpha t_{scaled}^{k_0} + (1 - \alpha) o_{scaled}^{k_0}, \tag{5.9}$$

where $t_{scaled}^{k_0} \in [0, 1]$ is the scaled computing time for solving the sub-MILP, $o_{scaled}^{k_0} \in [0, 1]$ is the scaled objective of sub-MILP, and α is a constant.

Given c^{k_0}, the label k_0^* can be defined as

$$k_0^* = \underset{k_0 \in \mathbb{Z}^+}{\operatorname{argmin}} \, c^{k_0}, \tag{5.10}$$

and is usually evaluated through black-box optimization methods: Given the time limit and a collection of training instances of interest, one evaluates the LB algorithm introduced in Section 5.3 with different values of k_0, and k_0^* is estimated by choosing the value with the best performance assessed by (5.9), which is typically the largest k_0 such that the resulting sub-MILP can still be solved to optimality within the time limit. Since the evaluation process for each instance is quite expensive, we propose to approximate it through regression.

Regression for learning k_0

With a collected dataset $\mathcal{D} = (\boldsymbol{s}_i, k_{0i}^*)_{i=1}^N$ with N instances, a regression task can be analyzed to learn a mapping from the state of the problem to the estimated k_0^*. The regression model $f_\theta(\boldsymbol{s})$ can be obtained by solving

$$\theta^* = \underset{\theta \in \Theta}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f_\theta(\boldsymbol{s}_i), k_{0i}^*), \tag{5.11}$$

where $\mathcal{L}(f_\theta(\boldsymbol{s}_i), k_{0i}^*)$ defines the loss function, a typical choice for regression task being the mean squared error.

The scaled regression task

Let $\boldsymbol{x}'$ be the optimal linear programming (LP) fractional solution without local branching constraint, and let k' be the value of the left-hand side of the local branching constraint evaluated using $\boldsymbol{x}'$. Specifically, k' is computed by

$$k' = \Delta(\boldsymbol{x}', \bar{\boldsymbol{x}}). \tag{5.12}$$

As discussed in [89], any $k \geq k'$ is likely to be useless as the LP solution after adding the LB constraint would be unchanged. Hence, k' provides an upper bound for k.

We can therefore parametrize k as

$$k = \phi \, k', \tag{5.13}$$

where $\phi \in (0, 1)$. Now, we define the regression task over a scaled space $\phi \in (0, 1)$ instead of directly over k.

Given k_0^* and k_0', the label is easily computed by

$$\phi_0^* = \frac{k_0^*}{k_0'}. \tag{5.14}$$

The regression problem reduces to

$$\theta^* = \underset{\theta \in \Theta}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f_\theta(\boldsymbol{s}_i), \phi_{0i}^*), \tag{5.15}$$

where $\mathcal{L}(f_\theta(\boldsymbol{s}_i), \phi_{0i}^*)$ defines the loss function.

MILP representation We represent the state $\boldsymbol{s}$ as a bipartite graph $(\mathbf{C}, \mathbf{E}, \mathbf{V})$ [27]. Given a MILP instance, let n be the number of variables with d features for each variable, m be the number of constraints with q features for each constraint. The variables of the MILP, with $\mathbf{V} \in \mathbb{R}^{n \times d}$ being their feature matrix, are represented on one side of the graph. On the other side are nodes corresponding to the constraints with $\mathbf{C} \in \mathbb{R}^{m \times q}$ being their feature matrix. A constraint node i and a variable node j are connected by an edge (i, j) if variable i appears in constraint j in the MILP model. Finally, $\mathbf{E} \in \mathbb{R}^{m \times n \times e}$ denotes the tensor of edge features, with e being the number of features for each edge.

Regression model Given that states are represented as graphs, with arbitrary size and topology, we propose to use *graph neural networks* (GNNs) [73, 74] to parameterize the regression model. Indeed, GNNs are size-and-order invariant to input data, i.e., they can process graphs of arbitrary size, and the ordering of the input elements is irrelevant. Another appealing property of GNNs is that they exploit the sparsity of the graph, which makes GNNs an efficient model for embedding MILP problems that are typically very sparse [27].

Our GNN architecture consists of three modules: the input module, the convolution module, and the output module. In the input layer, the state $\boldsymbol{s}$ is fed into the GNN model. The input module embeds the features of the state $\boldsymbol{s}$. The convolution module propagates the embedded features with graph convolution layers. In particular, our graph convolution layer applies the message passing operator, defined as

Algorithm 2: LB with scaled regression

Input: instance dataset $\mathcal{P} = \{\boldsymbol{p}_i\}_{i=1}^{M}$
for *instance* $\boldsymbol{p}_i \in \mathcal{P}$ **do**
  0. initialize the state $\boldsymbol{s}$ with an initial solution $\bar{\boldsymbol{x}}$;
  1. solve the LP relaxation and get solution $\boldsymbol{x'}$;
  2. compute $k' = \Delta(\boldsymbol{x'}, \bar{\boldsymbol{x}})$;
  3. predict $\phi_0 = f_\theta(\boldsymbol{s})$ by the regression model;
  4. compute $k_0 = \phi_0\, k'$;
  5. apply k_0 to execute the first LB iteration;
  6. update the incumbent $\bar{\boldsymbol{x}}$ and continue LB algorithm with its default setting;
  repeat
    execute the next LB iteration;
  until *termination condition is reached*;
end

$$\mathbf{v}_i^{(h)} = f_\theta^{(h)}\left(\mathbf{v}_i^{(h-1)}, \sum_{j \in \mathcal{N}(i)} g_\lambda^{(h)}\left(\mathbf{v}_i^{(h-1)}, \mathbf{v}_j^{(h-1)}, \mathbf{e}_{j,i}\right)\right), \tag{5.16}$$

where $\mathbf{v}_i^{(h-1)} \in \mathbb{R}^d$ denotes the feature vector of node i from layer $(h-1)$, $\mathbf{e}_{j,i} \in \mathbb{R}^m$ denotes the feature vector of edge (j,i) from node j to node i of layer $(h-1)$, and $\mathbf{f}_\theta^{(h)}$ and $\mathbf{g}_\lambda^{(h)}$ denote the embedding functions in layer h.

For a bipartite graph, a convolution layer is decomposed into two half-layers: one half-layer propagates messages from variable nodes to constraint nodes through edges, and the other one propagates messages from constraint nodes to variable nodes. The output module embeds the features extracted from the convolution module and then applies a pooling layer, which maps the graph representation into a single neuron. The output of this neuron is the prediction of ϕ_0.

LB with scaled regression

Our refined LB heuristic, *LB with scaled regression*, is obtained when k_0 is predicted by the regression model. The pseudocode of the algorithm is outlined in Algorithm 2.

5.4.2 Reinforced neighborhood search

In this section, we leverage reinforcement learning (RL) to adapt the neighborhood size iteratively. We first formulate the problem as a Markov Decision Process (MDP) [64]. Then,

we propose to use policy gradient methods to train a policy model.

Markov Decision Process

Given a MILP instance with an initial feasible solution, the procedure can be formulated as a MDP, wherein at each step, a neighborhood size is selected by a policy model and applied to run a LB iteration. The framework is shown in Figure 5.2. In principle, the state space $\mathcal{S}$ is the set of all the features of the MILP model and its solving statistics, which is combinatorial and arbitrarily large. To design an efficient RL framework for this problem, we choose a compact set of features from the solving statistics to construct the state. These features characterize the progress of the optimization process and are instance-independent, allowing broader generalization across instances.

For the action space State($\mathcal{A}$), instead of directly selecting a new k, we choose to adapt the value of k of the last LB iteration. The set of possible actions consists of four options

$$\{+k_{step}, 0, -k_{step}; \text{reset}\}, \tag{5.17}$$

where "$+k_{step}$"means increasing k by $k = k + k_{step}k$, "$-k_{step}$"means decreasing k by $k = k - k_{step}k$, "0"denotes keeping k without any change, and "reset"means resetting k to a default value. The policy π_k maps a state to one of the four actions. The step size k_{step} is a hyperparameter of the algorithm.

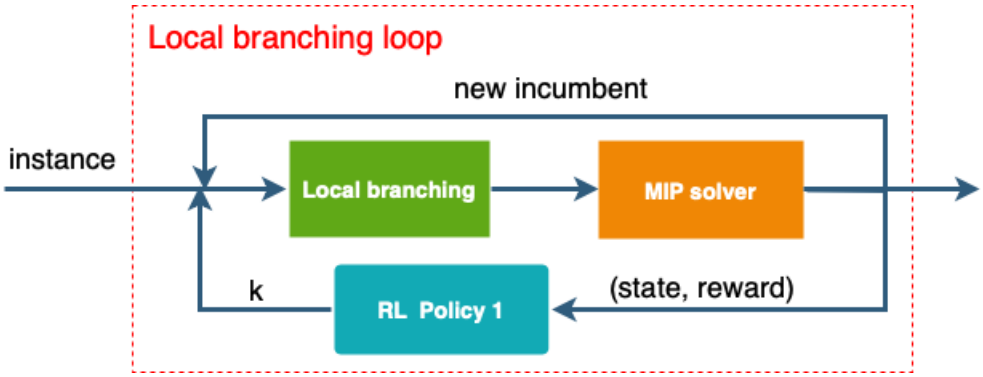

Figure 5.2 RL framework for adapting k

The compact description of states and actions offers several advantages. First of all, it simplifies the MDP formulation and makes the learning task easier. In addition, it allows for the use of simpler function approximators, which is critical for speeding up the learning process. In Section 5.5, we will show—by training a simple linear policy model using the

off-the-shelf policy gradient method—that the resulting policy can significantly improve the performance of the LB algorithm.

By applying the updated k, the next LB iteration is executed with time limit t_{limit}. Then, the solving sub-MILP statistics are collected to create the next state. In principle, the reward r_k is formulated according to the outcome of the last LB iteration, e.g., the computing time and the quality of the incumbent solution.

To maximize the objective improvement and minimize the solution time of the LB algorithm, we define the combinatorial reward as

$$r_k = o_{imp}(t_{max} - t_{elaps}), \tag{5.18}$$

where o_{imp} denotes the objective improvement obtained from the last LB iteration, t_{max} is the global time limit of the LB algorithm, and t_{elaps} is the cumulated running time.

The definition above is just one possibility to build a MDP for the LB heuristic. Actually, defining a compact MDP formulation is critical for constructing efficient RL algorithms for this problem.

Learning strategy

For training the policy model, we use the *Reinforce* policy gradient method [104], which allows a policy to be learned without any estimate of the value functions.

The refined LB heuristic, *reinforced neighborhood search for adapting k* is obtained when the neigborhood size k, is dynamically adapted by the RL policy. The pseudocode of the algorithm is outlined in Algorithm 3.

5.4.3 Further improvement by adapting LB node time limit

In the previous section, we suppose that the time limit for solving each local branching neighborhood is given. Indeed, the setting of time limit for each "LB node"decides how much computational effort is spent on each LB node, and therefore customizes how the overall time limit is split. E.g., a larger "LB node"time limit allows more computation for solving that LB node, which potentially leads to a larger k. However, it should be noted that the relation between the value of "LB node"time limit and the actual performance of LB scheme is not known. Therefore, we seek to address this issue by leveraging reinforcement learning again to learn policies for tailoring the time limit for each LB node. This new "RL-t"loop is built on top of the inner "RL-k"loop for adapting the LB neighborhood size. The scheme is shown in

Algorithm 3: Reinforced neighborhood search for adapting k

Input: instance dataset $\mathcal{P} = \{\boldsymbol{p}_i\}_{i=1}^{M}$
for *instance* $\boldsymbol{p}_i \in \mathcal{P}$ **do**
  0. initialize the state $\boldsymbol{s}$ with an initial solution $\bar{\boldsymbol{x}}$;
  1. compute k_0 by the procedure in Algorithm 1 or set k_0 by a default value;
  2. apply k_0 to execute the first LB iteration;
  3. collect the new state $\boldsymbol{s}$ and the incumbent $\bar{\boldsymbol{x}}$;
  repeat
    update k by policy $\pi_k(\boldsymbol{s})$;
    apply k and execute the next LB iteration;
    collect the new state $\boldsymbol{s}$ with the incumbent $\bar{\boldsymbol{x}}$;
  until *termination condition is reached*;
end

Figure 5.3.

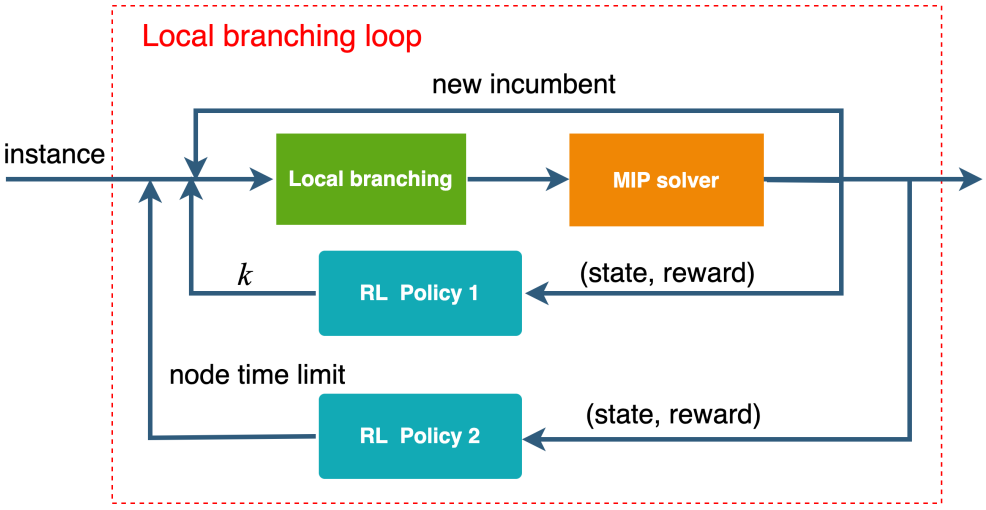

Figure 5.3 RL framework for adapting the time limit for solving the LB subproblem.

To design the action space for adapting the time limit of each LB node, we apply a similar setting as used for adapting k. The set of possible actions consists of four options

$$\{*t_{step}, 0, /t_{step}; \text{reset}\}, \tag{5.19}$$

where "$*t_{step}$"means increasing t by $t = t_{step}t$, "$/t_{step}$"means decreasing t by $t = \frac{t}{t_{step}}$, "0"denotes keeping t without any change, and "reset" means resetting t to a default value. The policy π_t maps a state to one of the four actions.

For the reward design of the "RL-t" loop, the reward also takes into account both the cost

of computation spent on solving the LB subproblem and the solution quality. Specifically, the reward r_t for the outer RL-t loop consists of two components. The first component r_1 maintains the same reward r_k as used in the "RL-k" loop, which attempts to maximize the improvement of the objective by the last LB iteration while minimizing the computing time. In addition, the LB environment explicitly returns a reward signal as a second component to penalize those LB iterations where the subproblem is too large to solve and there is no objective improvement within the time limit. The penalty reward r_p is a binary signal defined as

$$r_p = \begin{cases} 1 & \text{if the LB subproblem is not solved and no improving} \\ & \text{solution is returned,} \\ 0 & \text{otherwise.} \end{cases}$$

Formally, the reward r_t is a combination of two components, defined as

$$r_t = \beta_1 r_1 + \beta_2 r_2,$$

where $r_1 = r_k$, $r_2 = r_p$, $\beta_1 > 0$, $\beta_2 > 0$.

For training the policy $\pi_t(\boldsymbol{s})$, we use the same RL method as used for training the policy $\pi_k(\boldsymbol{s})$. Note that we fix the pretrained $\pi_k(\boldsymbol{s})$ policy while tuning $\pi_t(\boldsymbol{s})$ to make the learning process more stable.

A new LB alogorithm, *hybrid reinforced neighborhood search*, is obtained when the neigborhood size k and the time limit t for each LB node are dynamically adapted by the RL policies. The pseudocode of the algorithm is outlined in Algorithm 4.

5.5 Experiments

In this section, we present the details of our experimental results over five MILP benchmarks. We compare different settings of our approach against the original LB algorithm, using SCIP [105] as the underlying MILP solver.

5.5.1 Data collection

MILP instances

We evaluate on five MILP benchmarks: set covering (SC) [106], maximum independent set (MIS) [107], combinatorial auction (CA) [108], generalized independent set problem

Algorithm 4: Hybrid reinforced neighborhood search for adapting k and t

Input: instance dataset $\mathcal{P} = \{\boldsymbol{p}_i\}_{i=1}^{M}$
for *instance* $\boldsymbol{p}_i \in \mathcal{P}$ **do**
  0. initialize the state $\boldsymbol{s}$ with an initial solution $\bar{\boldsymbol{x}}$;
  1. compute k_0 by the procedure in Algorithm 1 or set k_0 by a default value;
  2. apply k_0 to execute the first LB iteration;
  3. collect the new state $\boldsymbol{s}$ and the incumbent $\bar{\boldsymbol{x}}$;
  repeat
    update k by policy $\pi_k(\boldsymbol{s})$;
    update t by policy $\pi_t(\boldsymbol{s})$;
    apply k, t and execute the next LB iteration;
    collect the new state $\boldsymbol{s}$ with the incumbent $\bar{\boldsymbol{x}}$;
  until *termination condition is reached*;
end

(GISP) [109, 110], and MIPLIB 2017 [111]. The first three benchmarks are used for both training and evaluation. For SC, we use instances with 5000 rows and 2000 columns. For MIS, we use instances on Barabási–Albert random graphs with 1000 nodes. For CA, we use instances with 4000 items and 2000 bids. In addition, to evaluate the generalization performance on larger instances, we also use a larger dataset of instances with doubled size for each benchmark, denoted by LCA, LMIS, LCA. The larger datasets are only used for evaluation.

For GISP, we use the public dataset from [95]. For MIPLIB, we select binary integer linear programming problems from MIPLIB 2017. Instances from GISP and MIPLIB are only used for evaluation.

For each instance, an initial feasible solution is required to run the LB heuristic. We use two initial incumbent solutions: (1) the first solution found by SCIP; (2) an intermediate solution found by SCIP, typically the best solution obtained by SCIP at the end of the root node computation, i.e., before branching.

Data Collection for regression

To collect data for the scaled regression task, one can use black-box optimization methods to produce the label ϕ_0^*. As the search space has only one dimension, we choose to use the grid search method. In particular, given a MILP instance, an initial incumbent $\bar{\boldsymbol{x}}$, the LP solution $\boldsymbol{x}'$, and a time limit for a LB iteration, we evaluate ϕ_0 from $(0, 1)$ with a resolution limit 0.01. For each ϕ_0, we compute the actual neighborhood size by $k_0 = k' \, \phi_0$, where $k' = \Delta(\boldsymbol{x}', \bar{\boldsymbol{x}})$. Then, k_0 is applied to execute an iteration of LB. From all the evaluated ϕ_0, the one with

best performance is chosen as a label ϕ_0^*.

The state $\boldsymbol{s}$ consists of context features of the MILP model and the incumbent solution. The state $\boldsymbol{s}$ together with the label k^* construct a valid data point $(\boldsymbol{s}, k^*)$.

5.5.2 Experimental setup

Datasets

For each reference set of SC, MIS and CA problems, we generate a dataset of 200 instances, and split each dataset into training (70%), validation (10%), and test (20%) sets. For larger instances, we generate 40 instances of LSC, LMIS and LCA problems, separately. The GISP dataset contains 35 instances. For MIPLIB, we select 29 binary MILPs that are also evaluated by the original LB heuristic [85].

Model architecture and feature design

For the regression task, we apply the GNNs described in the paper with three modules. For the input module, we apply 2-layer perceptron with the *rectified linear unit* (ReLU) activation function to embed the features of nodes. For the convolution module, we use two half-layers, one from nodes of variables to nodes of constraints, and the other one from nodes of constraints to nodes of variables. For the output module, we also apply 2-layer perceptron with the ReLU activation function. The pooling layer uses the *sigmoid* activation function. All the hidden layers have 64 neurons.

Given the input $x \in \mathbb{R}$, the Sigmoid and ReLU functions are defined as

$$\text{Sigmoid}(x) = \frac{\exp(x)}{\exp(x) + 1}, \tag{5.20}$$

$$\text{ReLU}(x) = \max(0, x). \tag{5.21}$$

For the bipartite graph representation, we reference the model used in [27]. The features in the bipartite graph are listed in Table 5.1. In practice, if the instance is a pure binary MILP, one can choose a more compact set of features to accelerate the training process. For example, the features describing the type and bound of the variables can be removed.

For the two RL policies, we apply a linear model with seven inputs and four outputs. The set of features used by the RL policies is listed in Table 5.2.

Table 5.1 Description of the features in the bipartite graph $\boldsymbol{s} = (\mathbf{C}, \mathbf{E}, \mathbf{V})$.

Tensor	Feature	Description
$\mathbf{C}$	bias	Bias value, normalized with constraint coefficients.
$\mathbf{E}$	coef	Constraint coefficient, normalized per constraint.
$\mathbf{V}$	coef	Objective coefficient, normalized.
	binary	Binary type binary indicator.
	integer	Integer type indicator.
	imp_integer	Implicit integer type indicator
	continuous	Continuous type indicator.
	has_lb	Lower bound indicator.
	has_ub	Upper bound indicator.
	lb	Lower bound.
	ub	Upper bound.
	sol_val	Solution value.

Table 5.2 Description of the input features of the RL policy.

Feature	Description
optimal	Sub-MILP is solved and the incumbent is updated, indicator.
infeasible	Sub-MILP is proven infeasible, indicator.
improved	Sub-MILP is not solved but the incumbent is updated, indicator.
not_improved	Sub-MILP is not solved and no solution is found, indicator.
diverse	No improved solution is found for two iterations, indicator.
t_available	time available before the time limit of the sub-MILP is reached
obj_improve	improvement of objective.

Training and evaluation

For the regression task, the model learns from the features of the MILP formulation and the incumbent solution. We use the mean squared error as the loss function. We train the regression model with two scenarios: the first one trains the model on the training set of SC, MIS and CA separately, the other one trains a single model on a mixed dataset of the three training sets. The models trained from the two scenarios are compared on the three test sets.

For the RL task, since we only use the instance-independent features selected from solving statistics, the RL policy is only trained on the training set of SC, and evaluated on all the test sets.

To further evaluate the generalization performance with respect to the instance size and the instance type, the RL policies (trained on the SC dataset) and the regression model (trained on the SC, MIS and LCS datasets) were evaluated on GISP and MIPLIB datasets.

Evaluation metrics

We use two measures to compare the performance of different heuristic algorithms. The first indicator is the *primal gap*. Let $\tilde{\boldsymbol{x}}$ be a feasible solution , and $\tilde{\boldsymbol{x}}_{opt}$ be the optimal (or best known) solution. The *primal gap* (in percentage) is defined as

$$\gamma(\tilde{\boldsymbol{x}}) = \frac{|c^T\tilde{\boldsymbol{x}}_{\text{opt}} - c^T\tilde{\boldsymbol{x}}|}{|c^T\tilde{\boldsymbol{x}}_{\text{opt}}|} \times 100,$$

where we assume the denominator is nonzero.

For the second measure, we use the *primal integral* [84], which takes into account both the quality of solutions and the solving time required to find them. To define the primal integral, we first consider a *primal gap function* $p(t)$ as a function of time, defined as

$$p(t) = \begin{cases} 1, & \text{if no incumbent until time } t, \\ \bar{\gamma}(\tilde{\boldsymbol{x}}(t)), & \text{otherwise,} \end{cases}$$

where $\tilde{\boldsymbol{x}}(t)$ is the incumbent solution at time t, and $\bar{\gamma}(\cdot) \in [0, 1]$ is the *scaled primal gap* defined by

$$\bar{\gamma}(\tilde{x}) = \begin{cases} 0, & \text{if } c^T\tilde{\boldsymbol{x}}_{\text{opt}} = c^T\tilde{\boldsymbol{x}} = 0, \\ 1, & \text{if } c^T\tilde{\boldsymbol{x}}_{\text{opt}} \cdot c^T\tilde{\boldsymbol{x}} < 0, \\ \frac{|c^T\tilde{\boldsymbol{x}}_{\text{opt}} - c^T\tilde{\boldsymbol{x}}|}{\max\{|c^T\tilde{\boldsymbol{x}}_{\text{opt}}|,\ |c^T\tilde{\boldsymbol{x}}|\}}, & \text{otherwise.} \end{cases}$$

Let $t_{\max} > 0$ be the time limit. The primal integral of a run is then defined as

$$P(t_{\max}) = \int_0^{t_{\max}} p(t)\, dt.$$

Details of experimental settings

To tune the learning rate for training the regression model, we have experimented different learning rates from 10^{-5} to 10^{-1} and have chosen a learning rate of 10^{-4}. We trained the model with a limit of 300 epochs.

For training the RL policies, we used the same method to tune the learning rate and have chosen a learning rate of 10^{-2} for π_k, and 10^{-1} for π_t . We trained the RL policies with a limit of 300 epochs.

For the hyperparameters, we have chosen $\alpha = 0.5$ as a trade-off between speed and quality for the cost metric for k_0. We used $\beta_1 = \beta_2 = 1$ for the two components of r_t. We used $k_{step} = 0.5$ for the action design in Section 5.4.2 and $t_{step} = 2$ for the action design in Section 5.4.3. We set a time limit of 10 seconds for each LB iteration for all the compared algorithms.

Our code is written in Python 3.7 and we use Pytorch 1.60 [112], Pytorch Geometric 1.7.0 [113], PySCIPOpt 3.1.1 [114], SCIP 7.01 [105] for developing our models and sovling MILPs.

5.5.3 Results

In order to validate our approach, we first implement LB as a heuristic search strategy for improving a certain incumbent solution using the MILP solver as a black-box.

Local branching search with adapting k

We perform the evaluations of our framework on the following four settings:

- *lb-sr*: Algorithm 2 with regression model trained by a homogenous dataset of *SC*, *MIS*, *CA*, separately;
- *lb-srm*: Algorithm 2 with regression model trained by a mixed dataset of *SC*, *MIS*, *CA*;
- *lb-rl*: Algorithm 3 with setting k_0 by a default value;
- *lb-srmrl*: combined algorithm using regression from Algorithm 2 (with regression model trained by mixed dataset of *SC*, *MIS*, *CA*) and RL from Algorithm 3.

We use the original local branching algorithm as the baseline. All the algorithms use SCIP as the underlying MILP solver and try to improve the initial incumbent with a time limit of $60s$. Our code and more details of the experiment environment are publicly available[1].

The evaluation results for the basic SC, MIS, CA datasets are shown in Table 5.3 and Table 5.4. Our first observation is that the learning based algorithms of our framework significantly outperform the original LB algorithm. Both the primal integral and the final primal gap of the four LB variants are smaller than those of the baseline over most datasets, showing improved heuristic behavior. Note that, although the regression model trained by using supervised learning and the policy model trained by RL can be used independently, they benefit from being used together. As a matter of fact, the hybrid algorithm *lb-srmrl* combining both methods achieves a solid further improvement and outperforms the other algorithms for most cases.

We also evaluate the impact of the choice of training set for the regression model. By comparing *lb-sr* and *lb-srm*, we observe that the regression model trained on a mixed dataset of *SC*, *MIS*, *CA* exhibit a performance very close to that of the model trained on a homogeneous dataset. Indeed, the GNN networks we used embed the features of the MILP problem and its incumbent solution. In particular, one significant difference between our method and those of previous works is that, instead of training a separate model for each class of instances, our method is able to train a single model yielding competitive generalization performances across instances. This is because our models predict the neighborhood size at a instance-wise level, rather than making predictions on variables.

Broader Generalization Next, we evaluate the generalization performance with respect to the size and the type of instances. Recall that our regression model is trained on a randomly mixed dataset of SC, MIS and CA problems, and the RL policy is only trained on the training set of SC. We evaluate the trained models on larger instances (LSC, LMIS, LCA) and new MILP problems (GISP, MIPLIB). The results of evaluation on larger instances are shown in Table 5.5 and Table 5.6, whereas the results on GISP and MIPLIB datasets, are shown in Table 5.7 and Table 5.8.

Overall, all of our learning-based LB algorithms outperform the baseline, and the hybrid algorithm *lb-srmrl* achieves the best performance on most datasets. These results show that our models, trained on smaller instances, generalize well both with respect to the instance size and, remarkably, across instances.

[1] `https://github.com/pandat8/ML4LB`

Table 5.3 Primal integral (geomeric means) for SC, MIS, CA problems.

	SC		MIS		CA	
Algo.	first	root	first	root	first	root
lb-base	57.013	5.697	5.713	4.893	9.296	3.308
lb-sr	3.643	1.751	1.537	3.499	6.358	2.025
lb-srm	4.338	1.818	1.419	3.654	6.556	2.120
lb-rl	17.304	4.513	2.616	2.616	4.450	2.042
lb-srmrl	**2.991**	**1.567**	**1.244**	**2.452**	**3.202**	**1.454**

Table 5.4 Final primal gap (geometric means in percentage) for SC, MIS, CA problems.

	SC		MIS		CA	
Algo.	first	root	first	root	first	root
lb-base	1411.624	1.390	0.326	**0.193**	3.307	1.218
lb-sr	1.425	1.102	**0.220**	0.210	2.460	0.877
lb-srm	1.645	1.118	0.247	0.232	2.466	0.971
lb-rl	6.876	1.621	0.449	0.263	0.558	**0.229**
lb-srmrl	**1.335**	**0.550**	0.275	0.242	**0.465**	0.311

Local branching search with adapting both k and t

In this section, we analyze the results of our approach on adapting both the neighborhood size k and time limit t for each LB node. Again, local branching is implemented as an independent heuristic search scheme for improving a certain incumbent solution using a black-box MILP solver.

We compare the *lb-base* and our best algorithm *lb-srmrl* outlined in the previous section with a new algorithm:

- *lb-srmrl-adapt-t*: Combined RL algorithm using regression from Algorithm 2 (with regression model trained by mixed dataset of *SC*, *MIS*, *CA*) and Hybrid RL from Algorithm 4.

In order to validate the effectiveness of our strategy for adapting t, we conducted the experiments on the two difficult MIP datasets (GISP, MIPLIB) with a longer global time limit, $600s$.

The results are shown in Table 5.9 and Table 5.10. In order to demonstrate the outcome of our LB algorithms on the solving progress of the instances over the running time, we plot the evolution of the average *primal integral* on the MIPLIB dataset in Figure 5.5. The plot on the left reports the results of the instances initialized by the first solution found by SCIP,

Table 5.5 Primal integral (geometric means) for LSC, LMIS, LCA problems.

	LSC		LMIS		LCA	
Algo.	first	root	first	root	first	root
lb-base	59.303	19.147	26.230	27.228	30.610	13.187
lb-srm	3.642	2.350	4.746	9.175	19.639	6.869
lb-rl	45.658	7.889	5.476	7.929	17.338	7.081
lb-srmrl	**2.790**	**1.876**	**1.819**	**5.807**	**10.078**	**4.311**

Table 5.6 Final primal gap (geometric means in percentage) for LSC, LMIS, LCA problems.

	LSC		LMIS		LCA	
Algo.	first	root	first	root	first	root
lb-base	8105.973	4.221	18.633	12.342	26.429	5.156
lb-srm	2.379	1.242	0.230	0.659	15.882	1.961
lb-rl	136.401	4.216	0.362	0.206	3.341	0.987
lb-srmrl	**1.326**	**0.777**	**0.195**	**0.205**	**2.007**	**0.152**

Table 5.7 Primal integral (geometric means) for GISP and MIPLIB problems.

	GISP		MIPLIB	
Algo.	first	root	first	root
lb-base	19.833	16.692	12.318	9.003
lb-srm	**13.359**	10.173	11.319	**6.863**
lb-rl	18.949	15.429	12.676	8.318
lb-srmrl	13.739	**9.865**	**10.172**	7.151

Table 5.8 Final primal gap (geometric means in percentage) for GISP and MIPLIB problems.

	GISP		MIPLIB	
Algo.	first	root	first	root
lb-base	22.244	17.171	44.794	11.893
lb-srm	**10.307**	7.704	32.061	**8.730**
lb-rl	20.030	14.402	42.676	11.869
lb-srmrl	11.641	**5.929**	**20.575**	9.696

whereas the plot on the right reports the results of the same instances initialized by the best solution obtained at the end of the root node of the B&B tree.

Table 5.9 Primal integral (geometric means) for GISP and MIPLIB problems with a time limit of 600s for each instance.

	GISP		MIPLIB	
Algo.	first	root	first	root
lb-base	116.882	101.013	59.916	35.747
lb-srm-rl	108.345	91.600	54.116	33.038
lb-srm-rl-adapt-t	**81.888**	**68.042**	**50.843**	**28.558**

Table 5.10 Final primal gap (geometric means in percentage) for GISP and MIPLIB problems with a time limit of 600s for each instance.

	GISP		MIPLIB	
Algo.	first	root	first	root
lb-base	12.440	11.039	36.860	**3.389**
lb-srm-rl	12.230	11.991	20.532	5.273
lb-srm-rl-adapt-t	**5.880**	**5.328**	**16.723**	4.396

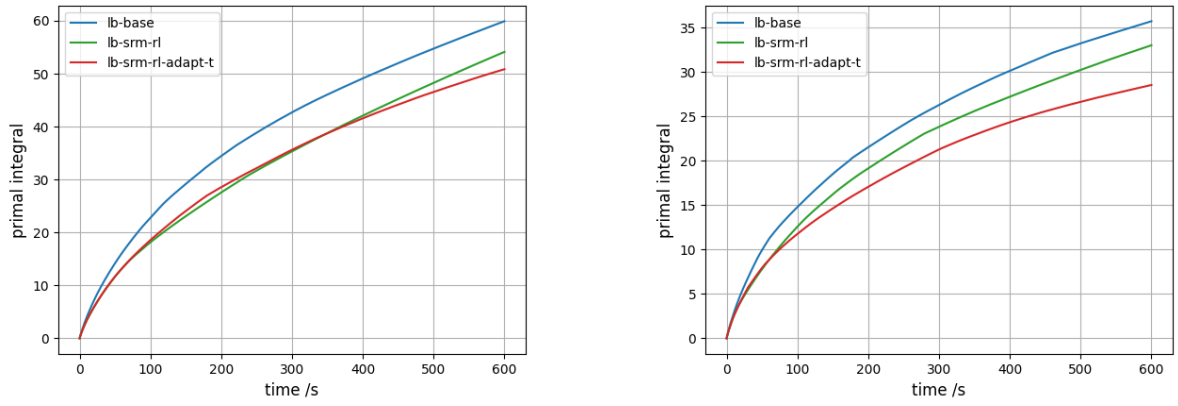

Figure 5.5 Evolution of the primal integral (geometric means) over time on binary MIPLIB dataset. Left / right: using the first / root solution to start LB.

From these results, we observe that: 1) All our learning-based LB algorithms converge faster than the LB baseline, showing improved heuristic behavior; 2) With our new strategy for adapting the LB node time limit t, the hybrid RL algorithm further improves its baseline with only a single RL policy for adapting k. The improvement becomes more and more significant as the solving time increases.

5.6 Local branching as a primal heuristic within a MILP solver

Local branching can also be implemented as a refinement heuristic within a generic MILP solver. In this section, we present two possible implementations of the LB algorithms outlined in the previous section, to be used as a primal heuristic. One major difference between these implementations and those of the previous section is how the global MILP search is structured. Instead of applying LB as a metaheuristic strategy, we use LB within the MILP solver to improve the incumbent at certain nodes of B&B tree. In particular, we considered the following two possible implementations:

- executing LB primal heuristic only at the root node: the MILP solver calls local branching only at the root node or at the node where the first incumbent solution is found;
- executing LB primal heuristic at multiple nodes: the MILP solver checks if there is a new incumbent or not for every f nodes in the B&B tree, and calls local branching if that is the case.

To configure the frequency f of executing the LB primal heuristic for the second implementation, we have conducted a simple hyperparameter search for f from the set of $\{1, 10, 100, 1000\}$, and the results showed that 100 performed the best. Thus, we set $f = 100$.

We evaluate the two primal heuristic implementations on the MIPLIB binary dataset and report the evolution of the average *primal integral* in Figure 5.6. Our observation is that the SCIP solver is improved by adding our learning based LB into SCIP as a primal heuristic. The results also suggest that the best strategy of executing LB is only to call it at the root node of the B&B tree (or at the node where the first incumbent solution is found).

5.7 Discussion

In this work, we have looked at the local branching paradigm by using a machine learning lens. We have considered the neighborhood size as a main factor for quantifying high-quality LB

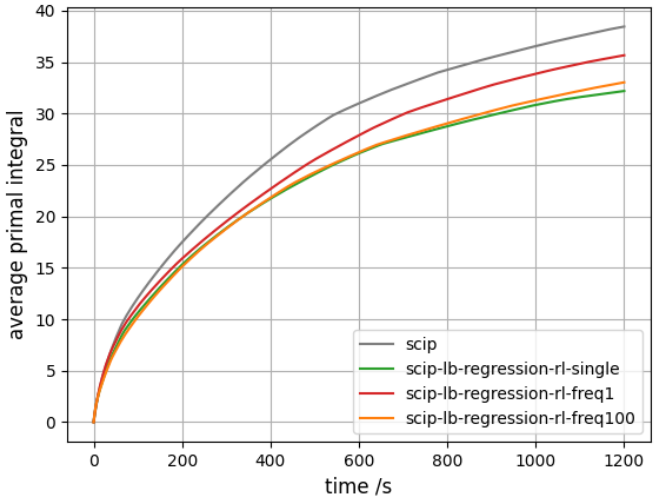

Figure 5.6 Evolution of primal integral (geometric means) over time on binary MIPLIB dataset (1200s).

neighborhoods. We have presented a learning based framework for predicting and adapting the neighborhood size for the LB heuristic. The framework consists of a two-phase strategy. For the first phase, a *scaled regression* model is trained to predict the size of the LB neighborhood at the first iteration through a regression task. In the second phase, we leverage reinforcement learning and devise a *reinforced neighborhood search* strategy to dynamically adapt the size at the subsequent iterations. Furthermore, we have also investigated the relation between "LB node" time limit t and the actual performance of LB scheme, and devised a strategy for adapting t. We have computationally shown that the neighborhood size and LB node time limit can indeed be learned, leading to improved performances and that the overall algorithm generalizes well both with respect to the instance size and, remarkably, across instances.

Our framework relies on the availability of an initial solution, thus it can be integrated with other refinement heuristics. For future research, it would be interesting to design more sophisticated hybrid frameworks that learn to optimize multiple refinement heuristics in a more collaborative way.

CHAPTER 6 LEARNING TO GENERATE NEIGHBORS IN METAHEURISTIC SEARCH

In this chapter, we further extend the idea in the previous chapter and present a methodology for integrating machine learning techniques into metaheuristics for solving combinatorial optimization problems. Namely, we propose a general machine learning framework for neighbor generation in metaheuristic search. In Section 6.3, we first define an efficient neighborhood structure constructed by applying a transformation to a selected subset of variables from the current solution. Then, the key of the proposed methodology is to generate promising neighborhoods by selecting a proper subset of variables that contains a descent of the objective in the solution space. To learn a good *variable selection* strategy, we formulate the problem as a *classification* task that exploits structural information from the characteristics of the problem and from high-quality solutions. In Section 6.4 and 6.5, we validate our methodology on two metaheuristic applications: a *Tabu Search* scheme for solving a Wireless Network Optimization problem and a *Large Neighborhood Search* heuristic for solving Mixed-Integer Programs. The experimental results show that our approach is able to achieve a satisfactory trade-off between the exploration of a larger solution space and the exploitation of high-quality solution regions on both applications.

6.1 Machine Learning for metaheuristics

Combinatorial Optimization (CO) is an important class of optimization problems in Operations Research (OR) and Computer Science (CS). In general, a CO problem is defined by a set of decision variables, a constrained solution space and an objective function. The goal of CO is to find optimal solutions with respect to the objective in the solution space.

Classical methods for solving CO problems can be roughly divided into three classes: exact methods, heuristics and metaheuristics. Mixed-Integer Programming (MIP) is one of the main paradigms of exact methods for modeling complex CO problems. Over the last decades, there has been increasing interest in improving the ability to solve MIPs effectively. Modern MIP solvers incorporate a variety of complex algorithmic techniques, such as primal heuristics [115], Branch and Bound (B&B) [1], cutting planes [2] and pre-processing, which results in complex and sophisticated software tools.

Exact methods are guaranteed to find optimal solutions as well as a proof of their optimality. On the one hand, due to the NP-hardness nature of many CO problems, solving them to

optimality within an exact algorithm is still a very challenging task. On the other hand, in many practical applications, one is more interested in getting a good solution within a reasonable time rather than finding an optimal one. Those practical requirements have motivated the development of *specific heuristics* and *metaheuristics* (MHs). Specific heuristics are usually designed for solving a specific type of CO problem, whereas MHs are frameworks for designing heuristics for solving general CO problems, and provide guidelines to integrate basic heuristic concepts with high-level diversification strategies.

It is worth noting that a lot of information can be produced and observed from MH processes, and therefore, a large volume of data can be collected. These data might provide valuable information about the optimization status of the process, the characteristics of the problem, the structures and properties of high-quality solutions in the solution regions being visited. However, such knowledge has not been fully exploited by traditional MH algorithms.

That can be viewed as a disadvantage when compared to application-targeted CO algorithms, in which exploiting the domain knowledge of the problem is generally preferable. Meanwhile, real-world CO problems have a rich structure. With similar instances repeatedly solved in many applications, statistical characteristics and patterns appear. This provides the opportunity for Machine Learning (ML) to extract structural properties of the problem from data and automatically produce learning-based MH strategies.

ML is a subfield of Artificial Intelligence (AI) that involves developing algorithms to learn knowledge from data and make predictions on new problems. In recent years, the application of ML techniques in CO became an emerging research area with quite a number of contributions for different purposes. On the one hand, some research has been devoted to develop heuristics for solving CO problems by ML, i.e., to perform "end-to-end learning" to directly generate good solutions for a CO problem. On the other hand, ML has been applied in combination with CO algorithms, where the learning based algorithms have the potential to achieve better performances, either because the current strategy of performing some auxiliary tasks is computationally expensive or because they are poorly understood from the mathematical viewpoint. For a detailed review of "learn to optimize", the interested reader is referred to the survey [16].

Specifically for metaheuristics, ML techniques can be used to infer patterns from data generated from MH processes. Integrating the extracted knowledge into the search strategies can lead MHs to search the solution space more efficiently and significantly improve the current performance. Recently, the application of ML techniques for MHs has attracted increasing research interest and we refer the interested reader to the following surveys [17,116].

In this chapter, we focus on the integration of ML techniques into MHs and propose a

general learning-based framework for neighbor generation in MHs. We first define an efficient neighborhood structure by applying a transformation to a selected subset of variables from the current solution. Then, the problem is to determine how to select a subset of variables that leads to a promising neighborhood of solutions containing a descent of the objective in the solution space. By conducting a classification task, our method learns good variable selection policies from both structural characteristics of the problem and high-quality solutions. We will demonstrate the effectiveness of our approach on two applications: a Tabu Search scheme for solving a Wireless Network Optimization problem and a Large Neighborhood Search heuristic for solving MIPs.

The remainder of the chapter is organized as follows. Section 6.2 introduces some relevant concepts. Section 6.3 delineates our proposed methodology. Section 6.4 and Section 6.5 present two detailed and successful applications of the proposed approach. Section 6.6 summarizes the paper and discusses future research.

6.2 Preliminaries

In this section, we introduce the necessary background and notation.

6.2.1 Combinatorial Optimization and metaheuristics

CO problems are a class of optimization problems with a set of decision variables and a defined solution space. Without loss of generality, a CO problem can be formulated into a constrained optimization problem as follows:

$$\begin{aligned}
\min \quad & c(\boldsymbol{x}) && (6.1)\\
\text{s.t.} \quad & g(\boldsymbol{x}) \leq \boldsymbol{b}, && (6.2)\\
& x_i \in \{0,1\},\ \forall i \in \mathcal{B}, && (6.3)\\
& x_j \in \mathbb{Z}^+,\ \forall j \in \mathcal{G}, && (6.4)\\
& x_k \geq 0,\ \forall k \in \mathcal{C}, && (6.5)
\end{aligned}$$

where the index set $\mathcal{V} := \{1, \ldots, n\}$ of decision variables is partitioned into $\mathcal{B}, \mathcal{G}, \mathcal{C}$, which are the index sets of binary, general integer and continuous variables, respectively.

Since many CO problems are NP-hard, determining optimal solutions by exact methods requires in the worst case exponential time and might be intractable, especially for large-size

applications. In many practical applications where the CO problems are hard and complex, practitioners are often interested in finding good-quality solutions in an "acceptable" amount of computing time rather than solving the problem to optimality. Therefore, heuristic algorithms are developed to compute high-quality solutions within a reasonable time.

Metaheuristics are general framework strategies for designing heuristics for solving CO problems, and provide guidelines to integrate basic heuristic schemes such as local search with high-level diversification strategies. A large part of metaheuristics are built on top of a basic *neighborhood search* (NS) scheme, where NS is a local search procedure that starts from an initial solution $\boldsymbol{x}$ and iteratively search for improving solutions by exploring a series of solution neighborhoods. At each NS iteration, the solution neighborhood is defined by a $Generate(\boldsymbol{x})$ function. A basic template for NS-based metaheuristic is shown in Algorithm 5.

In general, the structure of the solution neighborhoods and how the neighborhoods are explored, are designed according to the characteristics of the problem at hand. A neighborhood structure is typically defined by a move or transformation operator that applies a local transformation to the current solution and maps it to a subset of solutions. The resulting subset of solutions defines the "neighborhood" of the original solution.

6.2.2 Representation learning for CO

In ML, there is a vast library of models for representing CO problems depending on the format of input data of the CO task. For example, the model could be a linear function or some non-linear Artificial Neural Networks (ANNs) with a set of parameters to be optimized. In Deep Learning (DL), there are many types of neural networks and architectures available for modeling CO problems. For instance, a Multilayer Perceptron (MLP) is the simplest architecture of feedforward neural networks and can be used to model problems with fixed-size input data; Graph Neural Networks (GNNs) are developed to process data that are naturally representable with graphs; Recurrent Neural Networks (RNNs) can be applied to process CO problems with sequential data.

In particular, due to the ubiquity of graph data, problems over graphs arise in numerous application domains. Moreover, given the fact that the vast majority of CO problems have a discrete nature, many of them are naturally described in graphs or can be modeled into a graph structure. For instance, a network optimization problem can be naturally modeled by a graph and a generic MIP instance can also be represented into a bipartite graph [27]. These graphs have inherent structural commonalities and patterns, and there is a potential to exploit valuable graph features to learn patterns from data. Therefore, *graph representation learning* with the application of various GNNs [18, 73, 74, 117] has recently emerged as a

Algorithm 5: NS-based metaheuristic

Input: $\boldsymbol{x} = \boldsymbol{x}_0$
repeat
  $N(\boldsymbol{x}) \leftarrow Generate(\boldsymbol{x})$;
  $\boldsymbol{x}' \leftarrow \text{argmin}_{\boldsymbol{x}' \in N(\boldsymbol{x})} f(\boldsymbol{x}')$
  if $f(\boldsymbol{x}') < f(\boldsymbol{x}))$ **then**
    $\boldsymbol{x} \leftarrow \boldsymbol{x}'$;
  end
until *termination condition is reached*;
return $\boldsymbol{x}$

popular approach for studying CO with a machine learning perspective. Without loss of generality, GNNs learn a graph embedding with vertex representations based on the input features and the graph structure. In a nutshell, higher-level representations of a node are obtained by kernel convolutions which leverage its local structure.

In the literature, there has been an effort to learn algorithms for solving specific CO problems and many of them apply GNNs as the representation model. On the one hand, the first attempted paradigm is "end-to-end" learning for generating a heuristic solution by a ML model [20, 41, 43, 101, 118, 119]. However, these methods are typically limited to specific CO problems in which a heuristic solution can be easily constructed, and scaling to large-size instances is an issue. On the other hand, since a wide range of constrained CO problems can be formulated into a MIP model, there has also been increasing interest in learning decision rules to improve MIP algorithms [27, 28, 30, 92–94]. While it is shown that this direction has a great potential to improve the state-of-the-art of MIP algorithms, convincing generalization performances, and transfer learning across instances have not been fully tackled yet.

6.3 Methodology

In this section, we present our framework for learning to generate high-quality solution neighbors. In Section 6.3.1 and 6.3.2, we will first define the solution neighborhood and an efficient neighborhood structure by applying a transformation to a selected subset of variables from the current solution. Then, in Section 6.3.3 the problem becomes to determine how to select a subset of variables that leads to a promising solution neighborhood, i.e., one containing a descent point of the objective in the solution space. By conducting a classification task, our method learns promising variable selection policies from the structural characteristics of the problem and high-quality solutions.

6.3.1 Solution space and neighborhood structure

We consider an instance $p \in \mathcal{P}$ of a CO problem, where $\mathcal{X}$ is the solution space, i.e., the set of feasible solutions, and $f : \mathcal{X} \to \mathbb{R}$ is the objective function that maps a solution to its cost.

Definition 1. *Let $\boldsymbol{x}$ be a solution of an instance p such that $\boldsymbol{x} \in \mathcal{X}$. The neighborhood $N(\boldsymbol{x})$ of solution $\boldsymbol{x}$ is a subset of the solution space $\mathcal{X}$ defined by applying some local transformations to $\boldsymbol{x}$, i.e., $N(\boldsymbol{x}) \subseteq \mathcal{X}$.*

The definition of the neighborhood plays a critical role in the NS-based metaheuristics. It specifies how the metaheuristic search moves from the current solution over the solution space by an operator that applies a transformation of the current solution.

Definition 2. *Neighbor generation is defined by the transformation operator $\Delta : \mathcal{X} \to 2^{\mathcal{X}}$, which is a function (or a set of functions) that maps a solution $\boldsymbol{x}$ to a set of solutions $N(\boldsymbol{x}) \in \mathcal{X}$. If Δ is defined with a set M of parameters, the operator can be defined by $\Delta : \mathcal{X} \times M \to 2^{\mathcal{X}}$. A neighborhood $N(\boldsymbol{x}) \mapsto \Delta(\boldsymbol{x}, \boldsymbol{m})$ will be constructed by applying the transformation operator with $\boldsymbol{m} \in M$ to the current solution $\boldsymbol{x}$.*

The neighborhood structure is determined by the transformation operator, since the latter specifies how the current solution is perturbed and transformed to other solutions in the neighborhood. For instance, in a classical Traveling Salesman Problem (TSP) where a set of n cities and the distances between each pair of cities are given, the task is to find the shortest tour that visits each city exactly once. Given an arbitrary tour as the initial solution, a simple transformation operator can be defined by firstly removing k edges from the current tour and then adding k other edges to construct a new tour. This is known as the "k-OPT" neighborhood. For a generic CO problem at hand, there are many ways of defining a transformation operator. In general, two main aspects must be taken into consideration: *exploitation* (or *intensification*) and *exploration* (or *diversification*) of the solution space.

On the one hand, the smaller the perturbation induced by the operator, the closer the constructed neighborhood to the current solution. The metaheuristic search will thoroughly exploit the local solution regions around the current solution. On the other hand, with more perturbation or more randomness induced by the operator, the solution neighbors can be defined far from the current solution. The heuristic search will have more chances to explore more solution regions that have been less visited before. On the extreme case, when an operator completely perturbs the solution or the solution is allowed to be fully changed, the neighborhood could be expanded to the entire solution space, and the complexity of exploring this neighborhood will be very high, as high as solving the original problem. Instead, the

heuristic search will become totally random when the operator perturbs a part of the solution randomly. Moreover, the design of the transformation operator also strongly depends on the type of problem and its representation. The effectiveness of an operator might not be the same on different types of problems.

For neighbor generation, the main goal here is to design transformation operators with parameterizations that are able to construct high-quality solution neighborhoods (i.e., neighborhoods containing high-quality solutions), and achieve a good trade-off between exploitation and exploration of the solution space. As already mentioned, neither generating all possible neighbors nor constructing neighbors randomly generally leads to satisfactory performance. In order to guide the metaheuristic search to promising regions of the solution space, we will present a general ML framework for neighbor generation.

6.3.2 Variable selection for structural neighbor generation

In this section, we will define a neighborhood structure in NS-based metaheuristics. As mentioned before, we aim at generating promising solution neighbors by efficient transformation operators.

In NS, improving solutions are typically found from solution neighbors defined by perturbing only a part of the solution. In practice, in order to control the size of the neighborhood, the ratio of perturbation is typically set to a relatively small value, resulting in improving solutions that are only partially changed. Although this ratio can be increased by diversification strategies when no improving solution is found from the last search, it is still very common that the best solutions found by two consecutive local search iterations share a large part of solution with same values. Nevertheless, this "partial evolution" of the local optima induces a class of transformation operators for constructing structural neighborhoods, where the transformation is defined on a subset of variables.

Definition 3. *We define the "subset" transformation operator as* $\Delta : \mathcal{X} \times \mathcal{Y} \to 2^{\mathcal{X}}$*, where* $\mathcal{X}$ *denotes the solution space,* $\mathcal{V}_s$ *is a set of variables of interest, and* $\mathcal{Y} = \{0,\ 1\}^{|\mathcal{V}_s|}$ *defines a binary decision space for selecting a subset of variables from* $\mathcal{V}_s$*. Consequently,* $N(\boldsymbol{x}) \mapsto \Delta(\boldsymbol{x}, \boldsymbol{y})$ *with* $\boldsymbol{y} \in \mathcal{Y}$*.*

The definition of the *subset* transformation operator is highly dependent on the properties of the problem. Let $\overline{\mathcal{V}}_s = \{j \in \mathcal{V}_s : \bar{y}_j = 1\}$ be the support of $\mathcal{V}_s$. The variables in the complement set $\mathcal{V}_s \setminus \overline{\mathcal{V}}_s$ will be fixed to the values in the current solution, i.e., only the values of the variables in the binary support $\overline{\mathcal{V}}_s$ are allowed to change.

When the elements of $\boldsymbol{y}$ are all set to 1, all the variables in $\mathcal{V}_s$ will be allowed to change. As

discussed before, generating solution neighbors over the entire variable set $\mathcal{V}_s$ might result in a large local problem. On the other hand, the "partial evolution" of improving solutions in local search also indicates that it is possible to define efficient transformation operators only on a subset of variables. Moreover, common characteristics are often present in good solutions in many applications. For instance, in a classical TSP problem, the two cites that are far from each other are typically disconnected in good solutions, whereas the cites close to each other have a higher probability of being connected in a good solution. Hence, ML techniques can be employed to learn structural information from high-quality solutions and select a subset of variables that has a high probability of defining a solution neighborhood that contains a descent point of the objective in the solution space.

6.3.3 Learning a variable selection policy for structural neighbor generation

Given an instance of a generic CO problem with solution $\boldsymbol{x} \in \mathcal{X}$, we denote its current representation as $\boldsymbol{s} \in \mathcal{S}$, where $\mathcal{S}$ is the space of representations of the problem and consists of $\mathcal{P} \times \mathcal{X}$ and $\mathcal{P}$ denotes the parameter space of the CO problem. In order to generate a structural solution neighborhood with the subset transformation operator introduced in the previous section, an instantiation of $\mathcal{Y}$ is required to select a subset of variables from $\mathcal{V}_s$. Hence, a variable selection policy π for selecting $\boldsymbol{y} \in \mathcal{Y}$ can be defined by

$$\pi : \mathcal{S} \longrightarrow \mathfrak{P}(\mathcal{Y}) \tag{6.6}$$

$$\boldsymbol{s} \longmapsto \pi(\boldsymbol{s}), \tag{6.7}$$

where $\mathfrak{P}(\mathcal{Y})$ denotes the set of all probability distributions over $\mathcal{Y}$.

Now the question is:

> How to design a variable selection policy with respect to the instances of interest such that the resulting neighbor generation process is able to produce high-quality solution neighbors?

As mentioned before, the high-quality solutions found during the metaheuristic search often share common characteristics and patterns, and more importantly, many improved solutions in the neighborhood often share a part of the variables with the same values. Hence, we propose a general learning-based framework to extract these characteristics from data, and exploit the learned knowledge to guide the metaheuristic search towards compact and high-quality solution regions. Specifically, the framework learns a variable selection policy for neighbor

generation and the high-quality neighborhood structure will then be generated by selecting promising variables, the values of which are allowed to be changed from the current solution.

The pipeline of the framework consists of three blocks: *data generation*, *machine learning*, *neighbor generation design*. The framework is depicted in Figure 6.1.

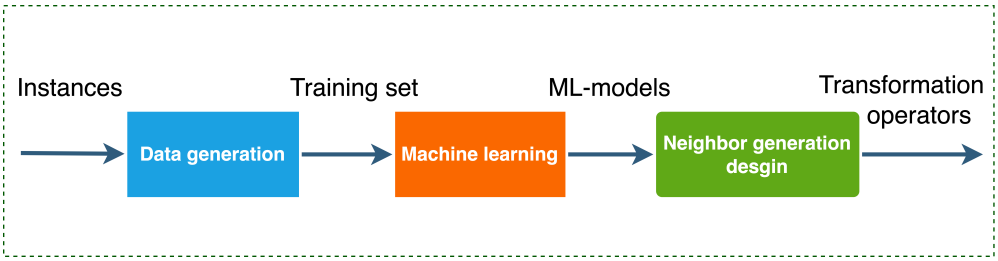

Figure 6.1 A learning-based framework for neighbor generation.

In the following, we will instantiate the learning problem for variable selection as a classification problem and train a variable selection policy through supervised learning. Within our method, the solution neighbors in metaheuristic search are generated based on our trained model.

Data generation

We aim to learn a policy π that maps the features $\boldsymbol{s}$ of a problem instance, to the labels $\boldsymbol{y}^*$, the binary classification decisions on the variables, for selecting the best subset of variables that leads to a successful change of the current solution. However, the label $\boldsymbol{y}^*$ is unknown, and we do not have any existing method to compute the exact $\boldsymbol{y}^*$. To estimate labels, we first apply a predefined expert algorithm for exploring improving subsets of variables in the predefined neighborhood, and take the best subset to compute $\boldsymbol{y}^*$.

Training the variable selection policy

Given a generated training data set $\mathcal{D}_{\text{train}} = \{(\boldsymbol{s}^{(j)}_{1:T_j}, \boldsymbol{y}^{*(j)}_{1:T_j}))\}_{j=1}^{M}$ of M instances ($j = 1, \ldots, M$) of a CO problem and the corresponding metaheuristic search trajectories (composed of T_j steps), the parameters θ^* of model π_{θ^*} can be obtained by solving the following classification problem:

$$\theta^* = \underset{\theta \in \Theta}{\operatorname{argmin}} \sum_{j=1}^{M} \sum_{i=1}^{T_j} \mathcal{L}\left(\pi_\theta(\boldsymbol{s}_i^{(j)}), \boldsymbol{y}_i^{*(j)}\right), \tag{6.8}$$

where $\{\boldsymbol{s}_i^{(j)}\}_{i=1}^{T_j}$ are the states of the j^{th} training CO instances, $\{\boldsymbol{y}_i^{*(j)}\}_{i=1}^{T_j}$ are the corresponding labels in a trajectory of T_j steps, and $\mathcal{L}\left(\pi_\theta(\boldsymbol{s}_i^{(j)}), \boldsymbol{y}_i^{*(j)}\right)$ denotes the loss function and will be

defined according to the CO application (see more details in Section 6.4.3, 6.5.2).

Modeling Given the fact that CO problems generally have a discrete nature and many of them can be represented by a graph structure, in the following, we will only introduce the GNN architectures [73, 74, 117]. Indeed, GNNs will be used in the two CO applications for validating our approach.

More precisely, if the solution state of a CO instance is modeled by a graph, a GNN can be implemented as the representation model. First of all, GNNs are size-and-order invariant to input data, i.e., they can process graphs of arbitrary size and topology, and the graph model is invariant to the ordering of the input elements, which brings a critical advantage compared to other neural networks such as RNNs. Another appealing property of GNNs is that they can exploit the sparsity of the graph, which makes GNNs an efficient model for embedding CO problems that are typically very sparse [27].

The basic architecture of GNNs consists of 3 modules: the input module, the convolution module, and the output module. In the input module/layer, the state $\boldsymbol{s}$ of the problem is fed into the GNN model. The input module embeds the features of $\boldsymbol{s}$. The convolution module propagates the features of the graph components by *graph convolution layers*. In particular, the architecture of the graph convolution layers used in this paper applies the message passing operator, defined as

$$\mathbf{v}_i^{(h)} = f_\theta^{(h)} \left(\mathbf{v}_i^{(h-1)}, \sum_{j \in \mathcal{N}(i)} g_\phi^{(h)} \left(\mathbf{v}_i^{(h-1)}, \mathbf{v}_j^{(h-1)}, \mathbf{e}_{j,i} \right) \right), \tag{6.9}$$

where $\mathbf{v}_i^{(h-1)} \in \mathbb{R}^d$ denotes the feature vector of node i from layer $(h-1)$, $\mathbf{e}_{j,i} \in \mathbb{R}^m$ denotes the feature vector of edge (j, i) from node j to node i, and $f_\theta^{(h)}$ and $g_\phi^{(h)}$ denote the embedding functions in layer h and are typically represented by neural networks, for example, MLP architectures.

Neighbor generation design

After training the classification model for variable selection, the next step is to apply the pretrained model for neighbor generation. It is important to note that, the trained classification model itself is probabilistic, and it maps the current solution state of an instance into a probability distribution in the binary decision space for each variable. One needs to select a strategy to make binary decisions on the variables.

The most straightforward way is to apply a *greedy* strategy. The decision is made by always

picking the class with a higher probability and the resulting strategy is *deterministic* and *greedy*. Another way is to sample decisions from the distribution, and hence the strategy is *probabilistic*. In general, the greedy strategy only selects the deterministic subset of variables and exploits the learned knowledge from training data. Instead, the probabilistic strategy selects from all the possible neighborhoods with a probability preference defined by the classification model, thus results in a better exploration of the less visited solution regions. There is no guarantee that one strategy is always better than the other. In practice, they can be combined. For each task, one should search for a good trade-off between exploitation of the local solution regions and exploration of the whole solution space.

Finally, the template for a NS-based metaheuristic guided by the variable selection policy π is given in Algorithm 6. The pretrained variable selection policy is integrated into the neighbor generation function to lead the heuristic search to promising solution regions.

6.4 Application 1: Tabu Search in Wireless Network Optimization

In this section, we apply our framework to the first case study, a Wireless Network Optimization (WNO) problem. We will demonstrate the effectiveness of our learning-based framework by generating solution neighbors in a Tabu Search scheme.

6.4.1 The tactical WNO problem

When telecommunications are necessary but standard networks are unavailable, as in the case of disaster relief operations, small temporary wireless networks, called tactical networks, are set up. These typically connect between 10 and 50 nodes (the key locations that must communicate) in a single network. The design of such networks can be optimized such that the network's weakest link is maximized. This, in turn, guarantees that all nodes can receive important information in a timely manner.

Tactical wireless network design is a complex non-linear combinatorial optimization problem that includes three sub-problems: the design of the topology (P_0), the configuration of the network (P_1), and the configuration of the antennas (P_2). These sub-problems are nested such that P_1 is defined given a topology, and P_2 is defined given a topology and a network configuration, namely

$$\underbrace{\max_{\text{topology } t \in T}}_{P_0} \quad \underbrace{\max_{\substack{\text{network} \\ \text{configuration}}}}_{P_1} \quad \underbrace{\max_{\substack{\text{antenna} \\ \text{configurations}}}}_{P_2} \quad o.$$

where T is the space of valid topologies and o is the objective function of the problem. The

Algorithm 6: NS-based metaheuristic guided by the variable selection policy

Input: $\boldsymbol{x} = \boldsymbol{x}_0$
repeat
  $N(\boldsymbol{x}) \leftarrow Generate(\boldsymbol{x}\ ;\ \pi)$;
  $\boldsymbol{x}' \leftarrow \text{argmin}_{\boldsymbol{x}' \in N(\boldsymbol{x})}\, f(\boldsymbol{x}')$;
  if $f(\boldsymbol{x}') < f(\boldsymbol{x}))$ **then**
    $\boldsymbol{x} \leftarrow \boldsymbol{x}'$;
  end
until *termination condition is reached*;
return $\boldsymbol{x}$

problem has been proposed by an industrial partner and the modeling of the full tactical wireless network design optimization problem can be found in [120].

Given a set of nodes V, the topology $t \in T$ can be any undirected tree (V, E) that describes how information travels in the network between every pair of nodes. Each edge in the topology represents a direct connection between two antennas in the network. A network configuration selects a root node (also known as a master hub or gateway) and assigns waveforms and channels/frequencies to the edges. The waveforms are the communication protocols that depend on the local structure of the edges and the channels and frequencies characterize their radio signals. Together with the antenna configurations, they determine which edges interfere with each other. An antenna configuration requires to define an angular alignment with respect to the azimuth as well as a set of activated beams, in the case of multi-beam antennas. Given all these properties, the radio signals can be physically modeled by also taking into account the path losses and fade margins of the terrain between every pair of nodes. The data transmission speed (called direct throughput) TP_{uv} for all the edges $[u, v] \in E$ can then be computed and, depending on the traffic scenario X and its distribution of congestion n_{uv}^X in the edges, their effective throughputs TP_{uv}/n_{uv}^X can also be computed. For the chosen traffic scenarios, the congestion of any directed edge (u, v) can be computed as a function of the number of descendants desc_v defined by the root node selected in the network configuration.

To evaluate a single topology $t \in T$, problem P_1, which is itself a complex combinatorial optimization problem, must be solved, and solving it requires solving P_2 many times as well. In turn, problem P_2 can be efficiently solved by a simple geometrically-based heuristic. Solving P_1 to optimality every time that a topology needs to be evaluated is too costly, especially in a NS context. It can be approximated in such a way that it can be solved efficiently by exhaustive enumeration. We denote the resulting approximated objective function by $\overline{f}(t)$. However, this approximation does not provide a feasible network because it does not define a proper frequency assignment. Alternatively, P_1 can also be tackled directly by considering greedy

frequency assignments which do provide feasible networks, although this takes considerably more time. We denote this final objective function by $f(t)$.

Given such procedures for solving P_1 and P_2, P_0, given by

$$\max_{t \in T} f(t),$$

can be solved with a NS-based MH in the space of topologies, where local directions of descent are computed with $\overline{f}(t)$.

6.4.2 Topology Tabu Search

Neighborhoods The neighborhood structure dictates how the local search can move in the space of solutions. Our neighborhood N is the edge-swap neighborhood. To construct the neighbor topologies, for each tree edge $e \in E$ in the current topology, we remove it, which disconnects the topology into two connected components, and we consider every possible way of reconnecting these two connected components with another edge.

An example of neighbor topologies is depicted in Figure 6.2, where the edges in blue were "swapped".

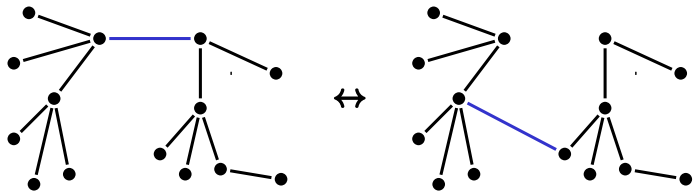

Figure 6.2 Neighbor Topologies.

Tabu Search Tabu Search (TS) is a MH that explores the solution space by temporarily not allowing to move in the direction from which it came. The only case in which this rule does not apply is when moving in such a direction improves the incumbent solution. This is done by using tabu lists that record the opposites of the last few moves, which then become tabu for the next few moves.

For our edge-swap neighborhood, two tabu lists are necessary: one for the dropped edges and one for the added edges. Therefore, a move, consisting of a dropped edge and an added edge, is tabu if either edges are in the corresponding tabu list. When a move is actually made to the current topology, the newly dropped edge is added to the tabu add list (it cannot be added again for the next few moves) and the newly added edge is added to the tabu drop list

(it can not be dropped again for the next few moves).

The lengths of the tabu lists determine for how many moves the dropped/added edges have tabu status. A reasonable length for the tabu drop list is

$$\left\lfloor \frac{1}{2}\sqrt{|V|-1} \right\rceil,$$

where $\lfloor x \rceil$ means x rounded to the nearest integer, and a reasonable length for the tabu add list is

$$\left\lfloor \sqrt{\frac{|V|\left(|V|-1\right)}{2}} \right\rceil.$$

The pseudocode for the resulting *topology Tabu Search* metaheuristic is given in Algorithm 7.

6.4.3 Learning to generate edge-swap neighbors for TS

As a NS-based MH, TS can be roughly described as a NS scheme plus a "tabu" strategy for preventing cycling by keeping a short-term memory of visited solutions stored in the tabu list. As discussed above, the topology design for the wireless network optimization problem can be solved by applying a topology TS algorithm (Algorithm 7). In this TS scheme, a solution neighbor of the current solution is constructed by applying an edge-swap move operator. Specifically, the move operator consists of two steps: an edge will be dropped from the current topology (by enforcing the value of the dropped edge variable from 1 to 0) and another edge will be added to complete a new topology (by enforcing the value of the added edge variable from 0 to 1). As a result, the edge-swap neighborhood consists of all the edge-swap moves.

In Algorithm 7, the neighborhood search at each TS iteration explores the entire edge-swap neighborhood by enumerating all the possible moves. Although the algorithm is guaranteed to find a local optimal solution by *exploiting* the entire edge-swap neighborhood, it is generally slow in terms of computing time since a complex subproblem (P_1) has to be solved to evaluate each "edge-swap" move, and another subproblem (P_2) has to be solved to evaluate each solution in P_1.

A potential improvement of the enumeration strategy is to apply a random sampling strategy to sample from droppable edge variables and addable edge variables, thus evaluating only a subset of possible moves. The random strategy generally *explores* a larger solution space than the enumeration strategy because, if the overall amount of time for the algorithm is fixed, it is able to do more iterations. However, random sampling might not be efficient enough for guiding the search towards high-quality solution neighbors. To achieve a better trade-off between exploitation (looking inside a neighborhood) and exploration (exploring

Algorithm 7: P_0 Topology Tabu Search

Input: $t \leftarrow t_0$, $t^* \leftarrow t_0$, $\overline{t}^* \leftarrow t_0$, $L \leftarrow ([\,], [\,])$
repeat
  $N(t) \leftarrow Generate(t)$;
  $t' \in \arg\max_{t' \in N(t)} \overline{f}(t')$ s.t. $x' \notin L$ or $\overline{f}(t') > \overline{f}(\overline{t}^*)$;
  if $\overline{f}(t') > \overline{f}(\overline{t}^*)$ **then**
    $\overline{t}^* \leftarrow t'$;
  end
  Make move $t \leftarrow t'$ and add the opposite move to L;
  if $f(t) > f(t^*)$ **then**
    $t^* \leftarrow t$;
  end
until *termination condition is reached*;
return t^*

more neighborhoods), we propose to exploit structural characteristics from improving moves, and learn good variable selection policies for selecting "drop" edge variables and "add" edge variables to generate size-reduced, but high-quality edge-swap neighborhoods. The scheme for variable selection consists of two components: a classifier for selecting the "drop" edge variable and another classifier for selecting the "add" edge classifier. The scheme is depicted in Figure 6.3.

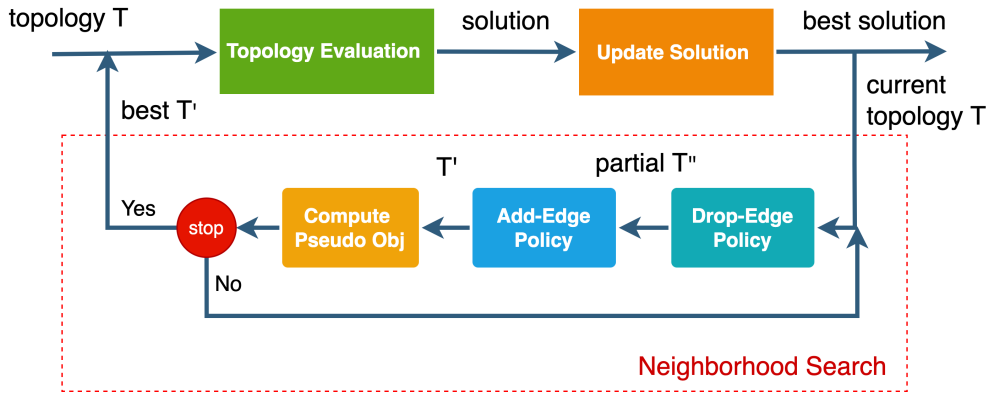

Figure 6.3 A learning-based framework for generating "edge-swap" neighbors.

Learning to drop edges

Let $\mathcal{S}$ denote the set of *states* of the current topology for an instance, and *action* $\mathcal{A}_d = \{0, 1\}^{|\mathcal{I}|}$ denotes the set of possible binary decisions for the droppable edge variables, where $\mathcal{I}$ is the

set of edges in the current topology. We aim to learn a "drop-edge" classifier $\pi^d : \mathcal{S} \to \mathcal{A}_d$ that maps $\boldsymbol{s}$, the state of the current topology, to the label $\boldsymbol{a}_d^*$, the binary classifications for selecting a subset of droppable edge variables that leads to a promising edge-swap move neighborhood. However, $\boldsymbol{a}_d^*$ is unknown, and we have to call the enumeration strategy to compute it. All the droppable edges that induce at least one improving edge-swap move will be included into the subset $\boldsymbol{a}_d^*$.

Given a training set $\mathcal{D}_{\text{train}} = \{(\boldsymbol{s}_{1:T_j}^{(j)}, \boldsymbol{a}_{d1:T_j}^{*(j)}))\}_{j=1}^{M}$ of M instances, the classifier π_θ^d with parameters θ, the best classifier on $\mathcal{D}_{\text{train}}$ can be obtained by solving the following problem:

$$\theta^* = \underset{\theta \in \Theta}{\operatorname{argmin}} \sum_{j=1}^{M} \sum_{i=1}^{T_j} \mathcal{L}\left(\pi_\theta^d(\boldsymbol{s}_i^{(j)}), \boldsymbol{a}_{di}^{*(j)}\right), \tag{6.10}$$

where $\{\boldsymbol{s}_i^{(j)}\}_{i=1}^{T_j}$ are the states of the j^{th} training WNO instance, $\{\boldsymbol{a}_{di}^{*(j)}\}_{i=1}^{T_j}$ are the corresponding labels in a trajectory of T_j steps, and $\mathcal{L}\left(\pi_\theta^d(\boldsymbol{s}_i^{(j)}), \boldsymbol{a}_{di}^{*(j)}\right)$ defines the loss function.

Feature design Since the topology naturally has a graph structure, we represent the state $\boldsymbol{s}$ of the current topology into a graph representation $(\mathbf{V}, \mathbf{E})$. Given a WNO instance, let n be the number of nodes with d features for each node. The matrix for the node features is denoted by $\mathbf{V} \in \mathbb{R}^{n \times d}$. Let e be the number of features for each edge, the tensor of edge features in the graph can be denoted by $\mathbf{E} \in \mathbb{R}^{n \times n \times e}$. The features used for learning to drop edges in the topology graph are listed in Table 6.1[1].

GNN model Again, due to intrinsic graph structure, it is natural to apply GNNs to model the drop-edge classifier. Our GNN architecture consists of 3 modules: the input module, the convolution module and the output module. The output module embeds the hidden features extracted from the convolution module and maps the embedding of each edge in the current topology into a two-neuron output.

Loss function From the data generation process, we observed that class distribution is unbalanced and only less than 50% of droppable edges from the current topology lead to an improving move, the remaining part of the edges are non-improving. Since the objective of the learning is to select as many "improving" edges to be dropped as possible, it is reasonable to put more effort in improving the predictions on the minority class. Therefore, we apply a *weighted* Cross Entropy (WCE) loss to train the model. Specifically, we add a penalty factor to impose a larger cost for making classification mistakes on the "improving" class during

[1]More detailed definitions of the listed features can be found in [120].

Table 6.1 Description of the features in the "droppable" graph $(\mathbf{V}, \mathbf{E})$.

Tensor	Feature	Description
$\mathbf{V}$	coordinates	X and Y coordinates in km
	desc_v	normalized $desc_v = ((\lvert desc_v \rvert - 1)/(\lvert V \rvert - 1))$
$\mathbf{E}$	pass_loss	path loss in dB
	fade_margin	fade margin in dB
	waveform	binary (point-to-point or point-to-multipoint)
	channel	frequency channel
	n_beams	number of beams used by the antenna of the predecessor
	tp_uv_a	throughput scenario A
	tp_uv_b	throughput scenario B
	tp_uv_c	throughput scenario C
	tp_uv_m	throughput by mixed scenarios of A, B, C
	edge indices	edge indices in current topology

training. Formally, the WCE loss is defined as

$$\mathcal{L}_{wce}\left(\hat{\boldsymbol{a}}, \boldsymbol{a}^*\right) = \frac{1}{|\mathcal{I}|} \sum_{i=1}^{|\mathcal{I}|} \left(\lambda a_i^* \log \hat{a}_i + (1-\lambda)(1-a_i^*) \log(1-\hat{a}_i)\right), \tag{6.11}$$

where $\hat{\boldsymbol{a}}$ denotes the prediction of probability given by the policy model, $\boldsymbol{a}^*$ denote the labels, $\mathcal{I}$ is the set of droppable edges in the current topology, $\lambda \in [0.5, 1]$ is the penalty parameter.

Learning to add edges

After dropping an edge, the next step is to select an edge to be added from all addable edges. Let $\mathcal{S}$ denote the set of *states* of the "addable" graph after dropping an edge. The "addable" graph we consider consists of all the nodes, the partial topology after dropping an edge, and plus the resulting add-able edges. Then, $\mathcal{A}_a = \{0, 1\}^{|\mathcal{I}|}$ denotes the set of binary decisions for addable edges, where $\mathcal{I}$ is the set of addable edges. We aim to learn an "add-edge" classifier $\pi^a : \mathcal{S} \to \mathcal{A}_a$ that maps $\boldsymbol{s}$, the state of the current topology, to $\boldsymbol{a}_a^*$, the binary decisions for selecting a subset of addable edges that lead to an improving edge-swap move. In order to compute $\boldsymbol{a}_a^*$, we call the enumeration strategy to evaluate all the addable edges and the edges that lead to an improving edge-swap move will be included into the subset $\boldsymbol{a}_a^*$.

Given a training set $\mathcal{D}_{\text{train}} = \{(\boldsymbol{s}^{(j)}_{1:T_j}, \boldsymbol{a}^{*(j)}_{a1:T_j}))\}_{j=1}^{M}$ of M instances, the parameters θ^* of the best classifier $\pi^a_{\theta^*}$ can be obtained by solving the following problem:

$$\theta^* = \underset{\theta \in \Theta}{\operatorname{argmin}} \sum_{j=1}^{M} \sum_{i=1}^{T_j} \mathcal{L}\left(\pi^a_\theta(\boldsymbol{s}^{(j)}_i), \boldsymbol{a}^{*(j)}_{ai}\right), \tag{6.12}$$

where $\{\boldsymbol{s}^{(j)}_i\}_{i=1}^{T_j}$ are the states of the j^{th} training WNO instance, $\{\boldsymbol{a}^{*(j)}_{ai}\}_{i=1}^{T_j}$ are the corresponding labels in a trajectory of T_j steps, and $\mathcal{L}\left(\pi^a_\theta(\boldsymbol{s}^{(j)}_i), \boldsymbol{a}^{*(j)}_{ai}\right)$ defines the loss function.

Feature design As modeling the graph for dropping edges, we model the state $\boldsymbol{s}$ of the state of the "addable" graph $(\mathbf{V}, \mathbf{E})$ into a graph representation. Given an instance, let n be the number of nodes with d features for each node. The matrix for the node features can be denoted by $\mathbf{V} \in \mathbb{R}^{n \times d}$. Let e be the number of features for each edge in the graph (including the edges in the current topology and all the addable edges), the tensor of edge features in the graph can be denoted by $\mathbf{E} \in \mathbb{R}^{n \times n \times e}$.

The features used for learning to add edges in the topology graph are listed in Table 6.2.

Table 6.2 Description of the features in the "addable" graph $(\mathbf{V}, \mathbf{E})$.

Tensor	Feature	Description
V	coordinates	X and Y coordinates in km
	desc_v	normalized $\lvert desc\_v\rvert = ((\lvert desc\_v\rvert - 1)/(\lvert V\rvert - 1))$
E	edge_type	binary indicator of edges in current topology or addable edges
	pass_loss	path loss in dB
	fade_margin	fade margin in dB
	edge indices	edge indices in current topology

GNN model As for the drop-edge classifier, we also apply GNNs to model the add-edge classifier for selecting the edge to be added. The architecture is the same as the drop-edge classifier.

Loss function The class distribution for addable edges is even more unbalanced than for the droppable case. Actually, only less than 10% of addable edges lead to an improving move. Therefore, we applied the WCE loss to train the model with a larger weight factor assigned to the minority class.

6.4.4 Numerical experiments

This section contains the experimental results for the WNO application. After presenting the data collection, we discuss the experimental setting and the evaluation metrics, respectively. Finally, the results are reported and discussed.

Data collection

Problem instance generation For the numerical experiments, as suggested by our industrial partner, the instances were generated in the following way. First, independent coordinates for the explicit nodes $v \in V$ are iteratively generated using

$$(x_v, y_v) = \left(\sqrt{U_0}\cos(2\pi U_1),\ \sqrt{U_2}\sin(2\pi U_3)\right) \tag{6.13}$$

where $U_0, U_1, U_2, U_3 \sim \mathcal{U}(0,1)$ are independent and identically distributed (IID). These coordinates are then scaled to match the average distance ratio of 10 km. This coordinate generation is repeated until the minimum distance is above 2 km and the maximum distance is below 150 km. Once this is achieved, for each possible edge $[u, v]$ with $u, v \in V$, a random uniform variable is sampled for the path loss and another is sampled for the fade margin, both according to empirical cumulative distribution functions derived from North-American datasets.

We generated small instances of 10 nodes, as well as larger instances of 30 nodes. For these instances, the TS MH is initialized with a minimum spanning tree with a specific distance based on the path losses and fade margins of the terrain between every pair of nodes [120].

Training data generation To collect data for training the drop-edge and add-edge classifiers, we call Algorithm 7 with the enumeration strategy to evaluate all possible moves in the edge-swap neighborhood, and compute labels according to the learning task. In particular, given a WNO instance, and an initialization of the topology, we execute Algorithm 7 as follows: for each TS iteration, we first call the TS algorithm with enumeration strategy to evaluate possible edge-swap moves. For each droppable edge, we evaluate all the addable edges. If the best resulting edge-swap move within the neighborhood of the droppable edge leads to a better approximated pseudo-objective, then the droppable edge is labeled as "improving"; otherwise, it is labeled as "non-improving". Given the dropped edge, the label of each addable edge is also decided by the quality of the resulting edge-swap move. If the move leads to a better approximated pseudo-objective, then the addable edge within the neigborhood of the corresponding dropped edge is labeled as "improving"; otherwise, it is

labeled as "non-improving". The features for each "droppable" graph and "addable" graph are also collected accordingly.

TS guided by GNNs A new TS algorithm, *topology TS with GNNs* (TS-GNN), is obtained when the GNN classifiers are applied to generate edge-swap neighborhoods in the topology Tabu Search. The pseudocode of the algorithm is outlined in Algorithm 8.

Experimental setting

Training We train the drop-edge policy and add-edge policy on the collected training dataset separately. The training dataset is generated from 30 small instances of 10 nodes. The collected data is split into training (70%), validation (10%), and test (20%) sets.

Evaluation We evaluated the compared algorithms listed in Section 6.4.4 on two evaluation sets. The first evaluation set contains 50 small instances of 10 nodes. In addition, to evaluate the generalization performance of our approach on larger instances, we also evaluate the model trained on small instances on a large evaluation set with 5 instances of 30 nodes.

Experimental environment Our code is written in Python 3.9 and we use Pytorch 1.60 [112], Pytorch Geometric 1.7.0 [113] for implementing and training the GNNs.

Evaluation metrics

We use the *primal integral* [84] to measure the performance of the compared algorithms. The *primal integral* was originally proposed to measure the performance of primal heuristics for solving mixed-integer programs. The metric takes into account both the quality of solutions and the computing time spent to find those solutions during the solving process. To define the primal integral, we first consider a *primal gap function* $p(t)$ as a function of time, defined as

$$p(t) = \begin{cases} 1, & \text{if no incumbent until time } t, \\ \bar{\gamma}(\tilde{\boldsymbol{x}}(t)), & \text{otherwise,} \end{cases}$$

where $\tilde{\boldsymbol{x}}(t)$ is the incumbent solution at time t, and $\bar{\gamma}(\cdot) \in [0, 1]$ is the *scaled primal gap*

$$\bar{\gamma}(\tilde{x}) = \begin{cases} 0, & \text{if } f(\tilde{\boldsymbol{x}}_{\text{opt}}) = f(\tilde{\boldsymbol{x}}) = 0, \\ 1, & \text{if } f(\tilde{\boldsymbol{x}}_{\text{opt}}) \cdot f(\tilde{\boldsymbol{x}}) < 0, \\ \frac{|f(\tilde{\boldsymbol{x}}_{\text{opt}}) - f(\tilde{\boldsymbol{x}})|}{\max\{|f(\tilde{\boldsymbol{x}}_{\text{opt}})|,\ |f(\tilde{\boldsymbol{x}})|\}}, & \text{otherwise,} \end{cases}$$

Algorithm 8: P_0 Topology TS with GNNs

Input: $t \leftarrow t_0$, $t^* \leftarrow t_0$, $\overline{t}^* \leftarrow t_0$, $\mathcal{L} \leftarrow ([\,], [\,])$
repeat
  $N(t) \leftarrow Generate(t\ ;\ \pi^d, \pi^a)$;
  $t' \in \arg\max_{t' \in N(t)} \ \overline{f}(t')$ s.t. $x' \notin \mathcal{L}$ or $\overline{f}(t') > \overline{f}(\overline{t}^*)$;
  if $\overline{f}(t') > \overline{f}(\overline{t}^*)$ **then**
    $\overline{t}^* \leftarrow t'$;
  end
  Make move $t \leftarrow t'$ and add the opposite move to $\mathcal{L}$;
  if $f(t) > f(t^*)$ **then**
    $t^* \leftarrow t$;
  end
until *termination condition is reached*;
return t^*

where $f(\tilde{\boldsymbol{x}})$ denotes the objective value given solution $\tilde{\boldsymbol{x}}$, and $\tilde{\boldsymbol{x}}_{\text{opt}}$ is either the optimal solution or the best one known for the instance.

Let $t_{\max} > 0$ be the time limit for executing the heuristic. The primal integral measure is then defined as

$$P(t_{\max}) = \int_0^{t_{\max}} p(t)\, dt.$$

Results

In order to validate our approach, we compared multiple versions of our *TS-GNN* algorithm with the baselines. Specifically, we compare the following algorithms:

- TS with *No-Classifier*, the baseline TS algorithm of topology TS with an enumeration strategy that evaluates all the possible edge-swap neighbors;
- TS with *Random-Add-Classifier*, the baseline TS algorithm with a sampling strategy that randomly selects edges to be added for generating edge-swap neighbors;
- TS with *Random-Add-Drop-Classifier*, the baseline TS algorithm with a sampling strategy that randomly selects both the edges to be dropped and the edges to be added for generating edge-swap neighbors;
- TS with *GNN-Add-Classifier*, the TS algorithm plus the GNN classifier for selecting the edges to be added for generating edge-swap neighbors;

- TS with *GNN-Drop-Classifier*, the TS algorithm plus the GNN classifier for selecting the edges to be dropped for generating edge-swap neighbors;
- TS with *GNN-Add-Drop-Classifier*, the TS algorithm plus the GNN classifiers both for selecting the edges to be dropped and for selecting the edges to be added.

We evaluated the performance of all the compared algorithms on both small instances with 10 nodes and larger instances with 30 nodes. For each test set, we computed the average primal integral defined in Section 6.4.4 as well as the average number of iterations. The primal integral is our main performance metric for evaluating metaheuristics and it measures the speed of convergence of the objective over the entire search time (the smaller, the better). In addition, the number of iterations counts the number of solution neighborhoods explored by the algorithm. With the same running time, it reflects how much exploration is guaranteed by modifications made to the basic TS scheme. Indeed, larger number of iterations for the same amount of time indicates a faster moving between solution neighborhoods and is generally preferable, although the outcome of the search still depends on the quality of the generated neighborhoods. The results are shown in Figures 6.5 and 6.7 for instances with 10 and 30 nodes, respectively.

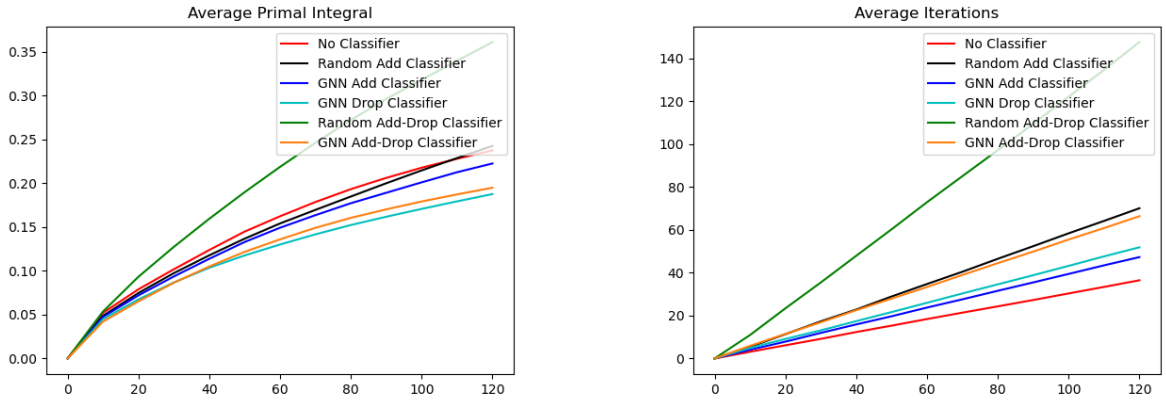

Figure 6.5 Evolution of the average primal integral and the average number of iterations over time on evaluation datasets for the instances of 10 nodes.

From these results, we observe that all our ML-based TS algorithms with GNN classifiers outperform the original TS baseline with no classifier. The *No-Classifier* baseline employs an enumeration strategy that evaluates all the possible moves in the edge-swap neighborhood. Although it is guaranteed to find the best move at each TS iteration, its more extensive exploitation results in the lowest number of iterations over all the compared algorithms.

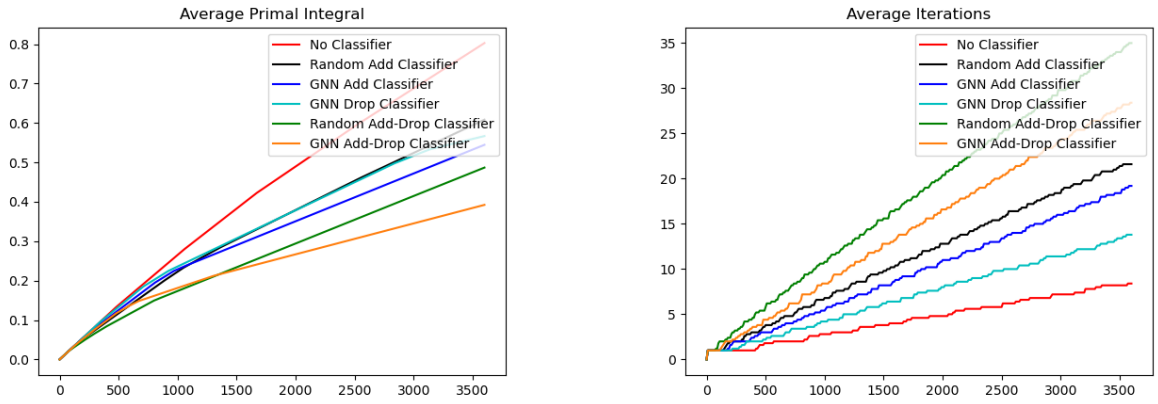

Figure 6.7 Evolution of the average primal integral and the average number of iterations over time on evaluation datasets for the instances of 30 nodes.

For small instances of 10 nodes, the performance of *No-Classifier* is still decent, as the solution space of these instances is small for exploration. However, it gives the worst results compared to all other algorithms on larger instances because the low number of iterations (slow exploitation) and the resulting lack of exploration becomes the main bottleneck of the algorithm.

The algorithms with random classifiers, on the contrary, are more efficient in terms of number of iterations. The *Random-Add-Classifier* algorithm explores a random subset of addable edges in the edge-swap neighborhood and the *Random Add-Drop Classifier* generates an even smaller neighborhood by sampling from both droppable and addable edges. On small instances with 10 nodes, *Random-Add-Classifier* achieves a better performance with a lower primal integral although the *Random-Add-Drop-Classifier* algorithm has a larger number of iterations. This is because, for small instances, the quality of the solutions in the sampled neighborhoods is more important than the number of iterations. Whereas for larger instances of 30 nodes, *Random-Add-Drop-Classifier* performs better than *Random-Add-Classifier* since the sampling efficiency becomes more important for exploring larger edge-swap neighborhoods.

On the one hand, the *No-Classifier* baseline can fully exploit each edge-swap neighborhood (*exploitation*) but has a low efficiency in terms of number of iterations. The baselines with random classifiers are more effective in increasing the number of iterations by reducing the size of each generated neighborhood (*exploration*), however, the quality of the generated neighborhood is restricted by its sampling efficiency. On the other hand, our complete ML-based algorithm with *GNN-Drop-Add-Classifier* achieves a smaller primal integral than all the baseline algorithms with a reasonably large number of iterations. Moreover, as the size

of instances increases, the *GNN-Drop-Add-Classifier* algorithm becomes more competitive and significantly outperforms all the compared algorithms. These results demonstrate that our approach offers a good trade-off in both exploration and exploitation of the solution space. Since the GNN classifiers are only trained with data generated from small instances, the results also show that our method generalizes well on larger instances.

6.5 Application 2: Large Neighborhood Search in MIP

In this section, we present how to apply our methodology to another application: a Large Neighborhood Search (LNS) heuristic for solving MIPs.

6.5.1 Large Neighborhood Search

LNS is a refinement heuristic, i.e., given an initial solution, it is applied to improve the solution by exploring a "large" solution neighborhood. There are several ways of describing a LNS scheme. We adopt the following simple one based of 3 building blocks:

- *destroy* function d: breaks a part of the current solution $\boldsymbol{x}$ and produces a solution neighborhood $N(\boldsymbol{x})$;
- *repair* function r: rebuilds the destroyed solution, typically by solving a sub-MIP defined by $N(\boldsymbol{x})$;[2]
- *accept* function a: decides whether the new solution should be accepted or rejected.

Given as an input a feasible solution $\bar{\boldsymbol{x}}$, it searches the best feasible solution in the neighborhood of $\bar{\boldsymbol{x}}$ (the size of the neighborhood is a parameter). Once the best feasible solution $\tilde{\boldsymbol{x}}$ in the neighborhood is found, the procedure updates $\bar{\boldsymbol{x}} = \tilde{\boldsymbol{x}}$. The method keeps searching for the best feasible solution in the new neighborhood until a stopping criterion is reached. The pseudocode of the scheme is given in Algorithm 9.

6.5.2 Learning to generate neighbors for LNS

In a LNS heuristic, the solution neighborhoods are generated by the destroy function that selects a subset of integer variables to be "destroyed", thus freed for neighbor generation. The remaining integer variables will be fixed to their values in the current solution. The classic LNS algorithm applies a randomized sampling strategy or hand-crafted rules for defining the

[2] In some cases, the repaired solution can be worse than the destroyed solution.

Algorithm 9: The LNS heuristic

Input: $\boldsymbol{x} = \boldsymbol{x}_0$; $\boldsymbol{x}^b = \boldsymbol{x}_0$
repeat
  $\boldsymbol{x}' = r(d(\boldsymbol{x}))$;
  if $a(\boldsymbol{x}',\ \boldsymbol{x})$ **then**
    $\boldsymbol{x} = \boldsymbol{x}'$;
  end
  if $f(\boldsymbol{x}') < f(\boldsymbol{x}))$ **then**
    $\boldsymbol{x} = \boldsymbol{x}'$;
  end
until *termination condition is reached*;
return $\boldsymbol{x}^b$

destroy function (see, e.g., [121]). However, our observation shows that only a small subset of variables have the potential to improve the current solution by changing their values and such a subset is strongly dependent to the structure of the problem. Hence, in order to optimize the performance of LNS, we aim at learning new variable selection strategies to select a subset of variables to be freed. In particular, we investigate the dependencies between the state of the problem, defined by a set of both static and dynamic features collected from the LNS procedure (e.g., context of the problem, incumbent solution) and the binary decisions about integer variables to be either freed or fixed.

Next, we will employ GNNs as the destroy policy and train GNNs through a classification task. Within our method, the LNS solution neighbors are generated based on our pretrained GNNs.

Let $\mathcal{S}$ denote the set of states of LNS for MIPs, and $\mathcal{A} = \{0,1\}^{|\mathcal{I}|}$ denote all candidate subsets of integer variables to be destroyed, where $\mathcal{I}$ is the set of integer variables. We aim to learn a destroy function $\pi : \mathcal{S} \to \mathcal{A}$ that maps $\boldsymbol{s}$, the features of a MIP instance, to the label $\boldsymbol{a}^*$, the action of best subset of variables to be destroyed. To estimate $\boldsymbol{a}^*$, we first apply a local branching heuristic [85] to search for improving subsets of variables and take the best found subset as the label.

Given a training set $\mathcal{D}_{\text{train}} = \{(\boldsymbol{s}^{(j)}_{1:T_j}, \boldsymbol{a}^{*(j)}_{1:T_j}))\}_{j=1}^{M}$ of M MIP instances and corresponding expert trajectories, the parameters θ^* of the best policy π_{θ^*} can be obtained by solving the following classification problem:

$$\theta^* = \underset{\theta \in \Theta}{\operatorname{argmin}} \sum_{j=1}^{M} \sum_{i=1}^{T_j} \mathcal{L}\left(\pi_\theta(\boldsymbol{s}_i^{(j)}), \boldsymbol{a}_i^{*(j)}\right), \tag{6.14}$$

where $\{\boldsymbol{s}_i^{(j)}\}_{i=1}^{T_j}$ are the states of the j^{th} MIP instance, $\{\boldsymbol{a}_i^{*(j)}\}_{i=1}^{T_j}$ are the corresponding labels in a trajectory of T_j steps, and $\mathcal{L}\left(\pi_\theta(\boldsymbol{s}_i^{(j)}), \boldsymbol{a}_i^{*(j)}\right)$ defines the loss function.

Feature design We represent the state $\boldsymbol{s}$ as a bipartite graph $(\mathbf{C}, \mathbf{E}, \mathbf{V})$ [27]. The variables of the MIP, with $\mathbf{V} \in \mathbb{R}^{n \times d}$ being their feature matrix, are represented on one side of the graph. On the other side are nodes corresponding to the constraints with $\mathbf{C} \in \mathbb{R}^{m \times q}$ being their feature matrix. A constraint node i and a variable node j are connected by an edge (i, j) if variable i appears in constraint j in the MIP model. Finally, $\mathbf{E} \in \mathbb{R}^{m \times n \times e}$ denotes the tensor of edge features, with e being the number of features for each edge. The features in the bipartite graph are listed in Table 6.3.

Table 6.3 Description of the features in the bipartite graph $\boldsymbol{s} = (\mathbf{C}, \mathbf{E}, \mathbf{V})$.

Tensor	Feature	Description
$\mathbf{C}$	bias	Bias value, normalized with constraint coefficients.
$\mathbf{E}$	coef	Constraint coefficient, normalized per constraint.
$\mathbf{V}$	sol_val	Solution value.

GNN model Given that the state of an MIP can be represented as a bipartite graph, we propose to use GNNs to parameterize the model for the destroy policy. Our GNN architecture also consists of 3 modules: the input module, the convolution module, and the output module. For a bipartite graph, a convolution layer is decomposed into two half-layers: one half-layer propagates messages from variable nodes to constraint nodes through edges, and the other one propagates messages from constraint nodes to variable nodes. We refer the reader to [27] for more details. The output module embeds the features extracted from the convolution module for the prediction of each variable, which maps the graph representation embedding of each variable into a two-neuron output.

Loss function The class distribution is highly unbalanced, e.g., for the generated training dataset, only less than 10% variables from the current solution are improving variables. In order to address the imbalanced distribution, we applied the WCE loss and focal loss [122] to train the model.

LNS guided by GNNs Our refined LNS heuristic, *Large Neighborhood Search with GNNs* (LNS-GNN), is obtained when the GNN model is employed as the destroy policy in LNS. The pseudocode of the algorithm is given by Algorithm 10.

Algorithm 10: The LNS-GNN heuristic

Input: $\boldsymbol{x} = \boldsymbol{x}_0$; $\boldsymbol{x}^b = \boldsymbol{x}_0$
repeat
  $\boldsymbol{x}' = r(d(\boldsymbol{x}\ ;\ \pi))$;
  if $(\boldsymbol{x}',\ \boldsymbol{x})$ **then**
    $\boldsymbol{x} = \boldsymbol{x}'$;
  end
  if $f(\boldsymbol{x}') < f(\boldsymbol{x}))$ **then**
    $\boldsymbol{x} = \boldsymbol{x}'$;
  end
until *termination condition is reached*;
return $\boldsymbol{x}^b$

6.5.3 Numerical experiments

In this section, we present the experimental results for the ML-based LNS. As for the WNO application, we present the data collection, discuss the experimental setting and and the results. The evaluation metrics remain the same as presented in Section 6.4.4.

Data collection

MIP benchmark We evaluate on 126 binary MIP instances from MIPLIB 2017 [111] for training and evaluation. For each instance, an initial feasible solution is required to start the LNS heuristic. We use an intermediate solution found by SCIP [105], typically the best solution obtained by SCIP at the end of the root node computation, i.e., before branching.

Training data generation To collect data for the classification task, we use an expert policy, Local Branching (LB) [85] to search over all the possible neighborhoods within the LNS search space, and return the best found solution to compute the labels of subsets of variables to be freed. In particular, given a MIP instance, an initial incumbent $\bar{\boldsymbol{x}}$, and a time limit for the LNS, we execute the LNS search as follows: for each LNS iteration, we first call the LB expert to explore all the LNS solution neighborhoods with a LB time limit. The best solution $\boldsymbol{x}^*$ found by LB is used to compare with $\bar{\boldsymbol{x}}$. As a result, the subset of variables with changed values are labeled as improving variables and are used to define the LNS subproblem. In order to collect multiple training data points on the same instance, a LNS trajectory is generated by recursively producing the next LNS subproblem with the labels and solving the LNS subproblem. The LNS subproblem is solved by calling the MIP solver and the incumbent solution is updated for the next LNS iteration.

At each LNS iteration, the state $\boldsymbol{s}$ consists of context features of the MIP model and the incumbent solution. The state $\boldsymbol{s}$ together with the label $\boldsymbol{a}^*$ define a valid data point $(\boldsymbol{s}, \boldsymbol{a}^*)$.

Experimental setting

Training In the classification task for producing a good destroy policy, the GNN model learns from the features of the MIP formulation and its incumbent solution. We train the GNNs on training data generated from 29 binary MIP instances from the MIPLIB dataset.

Evaluation We evaluated the compared algorithms on an evaluation set of the remaining 97 binary MIP instances from the MIPLIB dataset.

Experiment environment Our code is written in Python 3.7 and we use Pytorch 1.60 [112], Pytorch Geometric 1.7.0 [113], PySCIPOpt 3.1.1 [114], SCIP 7.01 [105] for developing our models and solving MIPs.

Results

We compare our *LNS-GNN* algorithm with the following baselines:

- *LNS-Random*, the LNS baseline algorithm;
- *LB*, the LB heuristic baseline that explores all the possible LNS neighborhoods within the same distance.

For measuring the heuristic performance of the compared algorithms, we also compute the average primal integral defined in Section 6.4.4. We ran the listed algorithms for 60 seconds on each instance. The results are shown in Figure 6.8.

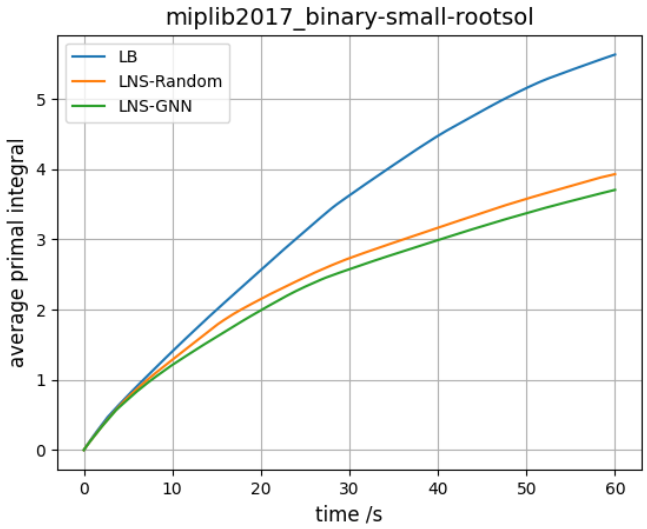

Figure 6.8 Evaluation results on MIPLIB binary dataset.

From the results in Figure 6.8, we can see that the primal integral of the *LB* baseline is the largest over the entire solving time, which indicates that exploring all the possible LNS solution neighborhoods within the same Hamming distance to the current solution is computationally expensive. Moreover, the *LNS-Random* baseline performs better than *LB* baseline by applying a random sampling strategy for selecting the subset of variables to be freed. Our *LNS-GNN* algorithm presents the best heuristic behavior in terms of the primal integral, showing that the pretrained GNN model is able to generate structural neighborhoods that contain improving solutions. These results demonstrate that our approach achieves a better trade-off between exploitation and exploration of the promising solution regions. Although, potentially, any of the baseline algorithms could be tuned to obtain better results, this is true for *LNS-GNN* too, and overall we believe that these results show clear promise.

6.6 Discussion

In this chapter, we presented a methodology for integrating ML techniques into metaheuristics for solving combinatorial optimization problems. Namely, we proposed a general ML framework for neighbor generation in metaheuristic search. We firstly defined a neighborhood structure constructed by applying a transformation to a selected subset of variables from the current solution. Then, the key of the proposed methodology is to generate promising neighborhoods by selecting a subset of variables that has a high probability of defining a solution neighborhood that contains a descent point of the objective in the solution space. We formulated the variable

selection problem as a classification problem that exploits structural information from the characteristics of the problem and high-quality solutions.

We demonstrated our methodology on two applications. The first problem we addressed occurs in the context of Wireless Network Optimization, where a TS metaheuristic is used for the topology design sub-problem. In a predefined topology neighborhood structure, we trained classification models to select sized-reduced, but high-quality solution neighborhoods. This allowed the TS metaheuristic to execute more iterations within the same amount of time. As each TS iteration requires to solve a series of complex combinatorial sub-problems, more iterations entail a greater exploration of the solution space. In addition, to demonstrate the broader applicability of our approach, we also applied our framework to the LNS framework for solving MIPs. The experimental results have shown that our approach is able to learn a satisfactory trade-off between the exploration of a larger solution space and the exploitation of promising solution regions on both applications.

Although deep neural networks such as GNNs have been wildly applied to represent combinatorial optimization problems, the current GNNs might not be expressive enough to capture all the crucial patterns from data [18]. For future research, it would be interesting to develop more expressive ML models that exhibit better transferability and scalability across broader classes of problems.

CHAPTER 7 GENERAL DISCUSSION

The three works presented in Chapters 4, 5, and 6 explore various ways of exploiting a problem's structure to improve the performances of a CO algorithm. Although each strategy operates at a different level of a different class of CO algorithm, from specific heuristics, primal heuristics in MILP to general metaheuristics, they share a number of motivations. This further discussion aims to highlight such connections and insights.

The most straightforward way to think of the application of ML for CO is to extract useful patterns from data and directly produce solutions for the underlying problem by "end-to-end" learning. In Chapter 4, we presented an imitation learning framework for producing constructive heuristics for a specific CO task, namely, learning elimination rules yielding high-quality chordal extensions. This work builds a complete pipeline for learning to generate algorithmic strategies for CO, from generation of synthetic problem, design of the ML task, features design, generation and labeling of training samples, training ML models and algorithmic design for integrating ML models in CO. The pipeline provides valuable concepts for exploring the research question and for designing the ML frameworks in Chapter 5 and 6. Moreover, while the current learning method is limited to an imitation setting, the MDP formulation of the problem opens promising scenarios for learning more sophisticated decision rules and addressing the performance of practical optimization algorithms, in conjunction with RL approaches.

Inspired by the application of ML in designing heuristics for a specific application in Chapter 4, we started to look deeper into more generic CO problems. In Chapter 5, we focused on MILP problems, and devised a ML-based framework for *Local Branching*, a well-known refinement primal heuristic. In this work, we studied how the size of the solution neighborhood can be quantified by learning from not only the characteristics of the problem and solutions, but also MILP solver's status. We applied both supervised learning and RL to extract valuable patterns from the solution neighborhoods as well as the optimization processes. This allows the learned policy to customize the size of the LB neighborhood with respect to both the MILP instance and the underlying MILP solver.

The ideas of exploiting critical properties of the solution neighborhoods from Chapter 5 stimulated further research in Chapter 6 towards designing a general ML methodological framework for learning neighborhood structures for generic NS based metaheuristics. From both the methodological and application point of view, each work builds on the insights learned from the previous to construct and extend the frameworks for the integration of ML

into CO.

The algorithmic design of how the ML are integrated in CO algorithms has been studied throughout the thesis. From an algorithmic standpoint, in all the three works, the ML strategy is called repeatedly as a building block in the optimization process. Although the ML policy in Chapter 4 acts as a heuristic to produce "end-to-end" solutions, the solutions are actually constructed step-by-step by the learned elimination rule. At each step, the ML model only needs to focus on the current partial solution and takes an action for the next step. This step-wise decision procedure for constructing a solution naturally provides the opportunity for ML to learn structural properties of the problem and solutions over the entire optimization process. In Chapter 5 and 6, the learned ML strategies are also integrated as algorithmic building block and called iteratively in the underlying optimization process. Here we highlight that this algorithmic structure with a tight combination of ML inference and optimization is the backbone of proposed CO algorithms in all the three contributions.

Feature design has played a critical role in the learning pipelines of all the three contributions. The informativeness of the input data with respect to the underlying ML task is important for the success of the following learning process. Indeed, the predictions of the ML model are inferred from the input data, and the performances of the ML model will be limited by the input data, regardless of learning method. Therefore, it is crucial to identify the useful features from the data that are correlated with the preferred predictions, and include them into the set of input features. Unlike other stand ML tasks with standard benchmarks of data, we have to carefully select features with respect to the learning task. The feature design is an important step in ML tasks for CO and it is generally engineered in conjunction with the design of the representation model and the training method. In general, a good trade-off between informativeness and conciseness is required for the learning process. On the one hand, the feature set must be informative enough for knowledge extraction and ML inference. On the other hand, the set needs to be concise and compact to control the size of the input the space. This is extremely important for conducting an efficient and stable learning process. In Chapter 5, we have designed a compact set of features for a RL task. Although training a policy with RL in the context of a CO task is considered hard and unstable in many cases, the design of the compact feature set enables us to implement an efficient policy model and also makes the training process more stable.

Another important aspect is the generalization scope. The generalization scope of the ML task for CO can be generally targeted into three classes. The first class aims to generalize to new instances of same size and of same type of problem, the second aims to generalize to larger instances of the same type, the last is the most ambitious, which aims to generalize to

instances from a different type of problem or from a heterogeneous data set. We also highlight that the feature design step is correlated with the generalization scope of the ML task. The common properties shared by the problems within the generalization scope should be carefully considered for feature design. For instance, when we trained the RL policies in Chapter 5, we have selected a set of features that only contain information collected from the solver's optimization process, hence, the features are independent from the class of problem. This allows us to design an efficient learning pipeline, that trains the policies with a generated synthetic homogeneous data set, and generalizes to a heterogeneous data set of real world problems.

CHAPTER 8 CONCLUSION AND RECOMMENDATIONS

In this thesis, we have investigated the application of ML techniques for designing CO algorithms. We have demonstrated the value of tight combination of ML and CO by three contributions. In this final chapter, we compile a brief summary of those contributions, and conclude by highlighting some limitations and identifying several avenues for future research on the topic.

8.1 Summary of works

Our first contribution, presented in Chapter 4, was motivated by the the early works using ML for CO. The most straightforward way to think of the application of ML for CO is to directly learn useful properties from data and produce solutions for the underlying problem by "end-to-end" learning. In Chapter 4, we have considered the computation of chordal extensions, a specific CO problem with a variety of applications in numerical optimization, and devised a framework for learning heuristic strategies yielding high-quality solutions for the problem. As a first building block of the learning framework, we have proposed an imitation learning scheme. Our developed ML model learns from the structure of the problem and the resulting ML strategy is sequentially called to construct a solution. The results have shown that our approach achieves remarkable generalization performance on graphs of larger size and from a different distribution. Another desirable behavior of our approach is that it allows to speed up the learning process by training on smaller synthetic problems by only paying a marginal loss of performance.

The initial work has led us to consider the integration of ML into a more general framework for solving CO problems. Our second contribution, presented in Chapter 5, addressed the algorithmic decisions within MILP algorithms. In particular, we focused on the Local Branching (LB), a well-known primal heuristic, and devised a ML-based framework for LB. In this work, we studied how the size of the solution neighborhood can be quantified by learning from not only the characteristics of the problem and its solutions, but also from the MILP solver's status. We applied both SL and RL to extract valuable patterns from the solution neighborhoods as well as the optimization processes. This allows the learned policy to customize the size of the LB neighborhood with respect to both the MILP instance and the underlying MILP solver. The resulting ML strategies are integrated into the LB algorithm and interact with the MILP solver at each iteration. Overall, we have computationally shown that the critical algorithmic decisions within a CO algorithm can indeed be learned by ML and the

resulting algorithm generalizes well both with respect to the instance size and, remarkably, across instances.

Finally, we presented in Chapter 6 a methodological contribution for integrating ML into metaheuristics for solving CO problems. Specifically, we proposed a general ML framework for the neighbor generation process in metaheuristic search. The key of the proposed methodology lies in the definition and generation of promising solution neighborhoods. The resulting classification framework exploits structural properties of both the problem and its high-quality solutions, and selects a subset of variables to define a promising neighborhood for metaheuristic search. We demonstrated the effectiveness of our framework on two metaheuristic schemes: *Tabu Search* and *Large Neighborhood Search*. The experimental results indicate that our approach is able to learn a satisfactory trade-off between the exploration of a larger solution space and the exploitation of promising local regions.

8.2 Limitations and future research

8.2.1 Modeling

Although deep neural networks such as GNNs have been wildly applied to represent CO problems, the current GNNs might not be expressive enough to capture all the crucial patterns from data [18]. For instance, we have tried to adapt the current architectures of GNNs to model another class of MIP problems, Mixed-Integer Quadratic Programs (MIQPs), and have trained the model to make algorithmic decisions for whether to solve the Semidefinite Program (SDP) relaxation at a B&B node. The training results have indicated that the model is not able to make predictions with enough accuracy. This might be because the current GNN architectures do not have enough representation capacity to model complex structures in the graphs, e.g., cliques.

For future research, it would be interesting to develop more expressive models that exhibit better transferability and scalability across broader classes of problems. For instance, enlarging the representation power of GNNs to represent hypernodes or to model multiple eliminations for the problem addressed in Chapter 4 will likely be key to handling practical tasks.

8.2.2 Multi-task learning

In this thesis, we have investigated different paradigms of integrating ML into CO algorithms. These methods show that various ML strategies can be learned from data and used to enhance the underlying optimization scheme. For instance, in Chapter 5, we have trained separate

decision policies for adapting different algorithmic parameters of the LB heuristic for MILP. Indeed, multiple algorithmic components within a MIP solver can be learned to improve the solver's performance, including but not limited to, primal heuristics, branching strategies, node selection and scaling strategies. This induces the question of multi-task learning and the performances of these ML tasks can be potentially improved by learning them jointly as partially done in [123].

Although the work of [124] has made one of the first steps to create a general environment for conducting ML tasks for MIP, for future research, it will be beneficial to develop a unified framework or an interface to coordinate these ML tasks for general CO problems in a more collaborative way.

REFERENCES

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

[2] R. Bixby and E. Rothberg, "Progress in computational mixed integer programming—a look back from the other side of the tipping point," *Annals of Operations Research*, vol. 149, no. 1, pp. 37–41, Feb 2007. [Online]. Available: https://doi.org/10.1007/s10479-006-0091-y

[3] T. Achterberg, "Scip: solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, 2009.

[4] T. Berthold, "Primal heuristics for mixed integer programs," Ph.D. dissertation, Zuse Institute Berlin (ZIB), 2006.

[5] G. Gamrath, T. Koch, A. Martin, M. Miltenberger, and D. Weninger, "Progress in presolving for mixed integer programming," *Mathematical Programming Computation*, vol. 7, no. 4, pp. 367–398, 2015.

[6] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, *Finding cuts in the TSP (A preliminary report)*. Citeseer, 1995, vol. 95.

[7] J. Patel and J. W. Chinneck, "Active-constraint variable ordering for faster feasibility of mixed integer linear programs," *Mathematical Programming*, vol. 110, no. 3, pp. 445–474, 2007.

[8] T. Achterberg and T. Berthold, "Hybrid branching," in *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 2009, pp. 309–311.

[9] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[10] R. S. Sutton and A. G. Barto, *Introduction to reinforcement learning*. MIT Press, Cambridge, 1998, vol. 135.

[11] K. A. Smith, "Neural networks for combinatorial optimization: a review of more than a decade of research," *INFORMS Journal on Computing*, vol. 11, no. 1, pp. 15–34, 1999.

[12] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[14] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[15] A. Lodi and G. Zarpellon, "On learning and branching: a survey," *Top*, vol. 25, no. 2, pp. 207–236, 2017.

[16] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," *European Journal of Operational Research*, vol. 290, no. 2, pp. 405–421, 2021.

[17] M. Karimi-Mamaghan, M. Mohammadi, P. Meyer, A. M. Karimi-Mamaghan, and E.-G. Talbi, "Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art," *European Journal of Operational Research*, vol. 296, no. 2, pp. 393–422, 2022.

[18] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković, "Combinatorial optimization and reasoning with graph neural networks," *arXiv preprint arXiv:2102.09544*, 2021.

[19] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 2692–2700.

[20] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2016.

[21] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning Combinatorial Optimization Algorithms over Graphs," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 6348–6358.

[22] W. Kool, H. Van Hoof, and M. Welling, "Attention, learn to solve routing problems!" *arXiv preprint arXiv:1803.08475*, 2018.

[23] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[25] E. B. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina, "Learning to branch in mixed integer programming," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[26] A. M. Alvarez, Q. Louveaux, and L. Wehenkel, "A machine learning-based approximation of strong branching," *INFORMS Journal on Computing*, vol. 29, no. 1, pp. 185–195, 2017.

[27] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi, "Exact combinatorial optimization with graph convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 15 554–15 566.

[28] H. He, H. Daume III, and J. M. Eisner, "Learning to search in branch and bound algorithms," *Advances in neural information processing systems*, vol. 27, pp. 3293–3301, 2014.

[29] Y. Tang, S. Agrawal, and Y. Faenza, "Reinforcement learning for integer programming: Learning to cut," in *International Conference on Machine Learning*. PMLR, 2020, pp. 9367–9376.

[30] E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao, "Learning to run heuristics in tree search," in *IJCAI*, 2017, pp. 659–666.

[31] G. Hendel, "Adaptive large neighborhood search for mixed integer programming," *Mathematical Programming Computation*, vol. 14, no. 2, pp. 185–221, 2022.

[32] T. Berthold and G. Hendel, "Learning to scale mixed-integer programs," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 5, 2021, pp. 3661–3668.

[33] P. Bonami, A. Lodi, and G. Zarpellon, "A classifier to decide on the linearization of mixed-integer quadratic problems in cplex," *Operations Research*, 2022.

[34] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Automated configuration of mixed integer programming solvers," in *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer, 2010, pp. 186–202.

[35] R. Moll, A. G. Barto, T. J. Perkins, and R. S. Sutton, "Learning instance-independent value functions to enhance local search," in *Advances in Neural Information Processing Systems*, 1999, pp. 1017–1023.

[36] J. A. Boyan and A. W. Moore, "Using prediction to improve combinatorial optimization search," in *Sixth International Workshop on Artificial Intelligence and Statistics*, 1997.

[37] R. Baltean-Lugojan, R. Misener, P. Bonami, and A. Tramontani, "Strong sparse cut selection via trained neural nets for quadratic semidefinite outer-approximations," Imperial College, London, Tech. Rep., 2018.

[38] E. Larsen, S. Lachapelle, Y. Bengio, E. Frejinger, S. Lacoste-Julien, and A. Lodi, "Predicting tactical solutions to operational planning problems under imperfect information," *arXiv preprint arXiv:1901.07935*, 2019.

[39] M. Fischetti, A. Lodi, and G. Zarpellon, "Learning milp resolution outcomes before reaching time-limit," in *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 2019, pp. 275–291.

[40] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 1263–1272.

[41] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác, "Reinforcement learning for solving the vehicle routing problem," *arXiv preprint arXiv:1802.04240*, 2018.

[42] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[43] D. Liu, A. Lodi, and M. Tanneau, "Learning chordal extensions," *Journal of Global Optimization*, vol. 81, no. 1, pp. 3–22, 2021.

[44] L. Vandenberghe, M. S. Andersen *et al.*, "Chordal graphs and semidefinite optimization," *Foundations and Trends® in Optimization*, vol. 1, no. 4, pp. 241–433, 2015.

[45] S. Wright, *Primal-Dual Interior-Point Methods*. Society for Industrial and Applied Mathematics, 1997. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9781611971453

[46] F. V. Fomin, G. Philip, and Y. Villanger, "Minimum fill-in of sparse graphs: Kernelization and approximation," *Algorithmica*, vol. 71, no. 1, pp. 1–20, Jan 2015.

[47] R. E. Bixby, "Solving real-world linear programs: A decade and more of progress," *Operations Research*, vol. 50, no. 1, pp. 3–15, 2002.

[48] E. Rothberg and B. Hendrickson, "Sparse matrix ordering methods for interior point linear programming," *INFORMS Journal on Computing*, vol. 10, no. 1, pp. 107–113, 1998.

[49] J. Agler, W. Helton, S. McCullough, and L. Rodman, "Positive semidefinite matrices with a given sparsity pattern," *Linear Algebra and its Applications*, vol. 107, pp. 101 – 149, 1988.

[50] M. Fukuda, M. Kojima, K. Murota, and K. Nakata, "Exploiting sparsity in semidefinite programming via matrix completion I: General framework," *SIAM Journal on Optimization*, vol. 11, no. 3, pp. 647–674, 2001.

[51] K. Nakata, K. Fujisawa, M. Fukuda, M. Kojima, and K. Murota, "Exploiting sparsity in semidefinite programming via matrix completion II: Implementation and numerical results," *Mathematical Programming*, vol. 95, no. 2, pp. 303–327, 2003.

[52] A. Majumdar, G. Hall, and A. A. Ahmadi, "A survey of recent scalability improvements for semidefinite programming with applications in machine learning, control, and robotics," *arXiv preprint arXiv:1908.05209*, 2019.

[53] Y. Zheng, G. Fantuzzi, A. Papachristodoulou, P. Goulart, and A. Wynn, "Chordal decomposition in operator-splitting methods for sparse semidefinite programs," *Mathematical Programming*, Feb 2019.

[54] Y. Zheng and G. Fantuzzi, "Sum-of-squares chordal decomposition of polynomial matrix inequalities," *arXiv:2007.11410 [cs, eess, math]*, Jul. 2020, arXiv: 2007.11410. [Online]. Available: http://arxiv.org/abs/2007.11410

[55] D. Bergman, C. H. Cardonha, A. A. Cire, and A. U. Raghunathan, "On the minimum chordal completion polytope," *Operations Research*, vol. 67, no. 2, pp. 532–547, 2019.

[56] M. Yannakakis, "Computing the minimum fill-in is np-complete," *SIAM Journal on Algebraic Discrete Methods*, vol. 2, no. 1, pp. 77–79, 1981. [Online]. Available: https://doi.org/10.1137/0602010

[57] A. George and J. W. Liu, "The evolution of the minimum degree ordering algorithm," *SIAM Review*, vol. 31, no. 1, pp. 1–19, 1989. [Online]. Available: https://doi.org/10.1137/1031001

[58] A. George, "Nested dissection of a regular finite element mesh," *SIAM Journal on Numerical Analysis*, vol. 10, no. 2, pp. 345–363, 1973.

[59] J. W. Liu, "The role of elimination trees in sparse factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 1, pp. 134–172, 1990.

[60] A. Guermouche, J.-Y. L'Excellent, and G. Utard, "Impact of reordering on the memory of a multifrontal solver," *Parallel Computing*, vol. 29, no. 9, pp. 1191 – 1218, 2003, parallel Matrix Algorithms and Applications.

[61] R. J. Vanderbei, "LOQO:an interior point code for quadratic programming," *Optimization Methods and Software*, vol. 11, no. 1-4, pp. 451–484, Jan. 1999.

[62] M. Garstka, M. Cannon, and P. Goulart, "A clique graph based merging strategy for decomposable sdps," 2019.

[63] J. Sliwak, M. Anjos, L. Létocart, J. Maeght, and E. Traversi, "Improving clique decompositions of semidefinite relaxations for optimal power flow problems," EasyChair Preprint no. 2546, EasyChair, 2020.

[64] R. A. Howard, *Dynamic programming and markov processes.* MIT Press, Cambridge, 1960.

[65] J. Bagnell, J. Chestnutt, D. M. Bradley, and N. D. Ratliff, "Boosting structured prediction for imitation learning," in *Advances in Neural Information Processing Systems*, 2007, pp. 1153–1160.

[66] N. D. Ratliff, D. Silver, and J. A. Bagnell, "Learning to search: Functional gradient techniques for imitation learning," *Autonomous Robots*, vol. 27, no. 1, pp. 25–53, 2009.

[67] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 627–635.

[68] S. Schaal, "Is imitation learning the route to humanoid robots?" *Trends in cognitive sciences*, vol. 3, no. 6, pp. 233–242, 1999.

[69] Y. Pan, C.-A. Cheng, K. Saigol, K. Lee, X. Yan, E. Theodorou, and B. Boots, "Agile autonomous driving using end-to-end deep imitation learning," in *Robotics: science and systems*, 2018.

[70] S. Ross and D. Bagnell, "Efficient reductions for imitation learning," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 661–668.

[71] D. Silver, J. Bagnell, and A. Stentz, "High performance outdoor navigation from overhead data using imitation learning," *Robotics: Science and Systems IV, Zurich, Switzerland*, 2008.

[72] H. M. Markowitz, "The elimination form of the inverse and its application to linear programming," *Management Science*, vol. 3, no. 3, pp. 255–269, 1957.

[73] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 2. IEEE, 2005, pp. 729–734.

[74] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *arXiv preprint arXiv:1709.05584*, 2017.

[75] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Advances in neural information processing systems*, 2015, pp. 2224–2232.

[76] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[77] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, 2017, pp. 1024–1034.

[78] Z. Li, Q. Chen, and V. Koltun, "Combinatorial optimization with graph convolutional networks and guided tree search," in *Advances in Neural Information Processing Systems*, 2018, pp. 539–548.

[79] S. Kullback, *Information theory and statistics*. Courier Corporation, 1997.

[80] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[81] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[82] D. J. Rose, "A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations," in *Graph theory and computing*. Elsevier, 1972, pp. 183–217.

[83] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 9244–9255.

[84] T. Berthold, "Measuring the impact of primal heuristics," *Operations Research Letters*, vol. 41, no. 6, pp. 611–614, 2013.

[85] M. Fischetti and A. Lodi, "Local branching," *Mathematical programming*, vol. 98, no. 1-3, pp. 23–47, 2003.

[86] M. Fischetti, F. Glover, and A. Lodi, "The feasibility pump," *Mathematical Programming*, vol. 104, no. 1, pp. 91–104, 2005.

[87] E. Danna, E. Rothberg, and C. L. Pape, "Exploring relaxation induced neighborhoods to improve mip solutions," *Mathematical Programming*, vol. 102, no. 1, pp. 71–90, 2005.

[88] T. Berthold, "Rens," *Mathematical Programming Computation*, vol. 6, no. 1, pp. 33–54, 2014.

[89] M. Fischetti and M. Monaci, "Proximity search for 0-1 mixed-integer convex programming," *Journal of Heuristics*, vol. 20, no. 6, pp. 709–731, 2014.

[90] M. Gendreau and J.-Y. Potvin, *Handbook of metaheuristics*. Springer, 2010, vol. 2.

[91] M. Fischetti and A. Lodi, "Heuristics in mixed integer programming," *Wiley Encyclopedia of Operations Research and Management Science*, 2010.

[92] D. Liu, M. Fischetti, and A. Lodi, "Learning to search in local branching," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 4, pp. 3796–3803, Jun. 2022. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/20294

[93] E. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina, "Learning to branch in mixed integer programming," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, Feb. 2016. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/10080

[94] M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik, "Learning to branch," in *International conference on machine learning*. PMLR, 2018, pp. 344–353.

[95] A. Chmiela, E. Khalil, A. Gleixner, A. Lodi, and S. Pokutta, "Learning to schedule heuristics in branch-and-bound," *arXiv preprint arXiv:2103.10294*, 2021.

[96] J.-Y. Ding, C. Zhang, L. Shen, S. Li, B. Wang, Y. Xu, and L. Song, "Accelerating primal solution findings for mixed integer programs based on solution prediction," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 02, pp. 1452–1459, 2020.

[97] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, and P. Wang, "Solving mixed integer programs using neural networks," *arXiv preprint arXiv:2012.13349*, 2020.

[98] N. Sonnerat, P. Wang, I. Ktena, S. Bartunov, and V. Nair, "Learning a large neighborhood search algorithm for mixed integer programs," *arXiv preprint arXiv:2107.10201*, 2021.

[99] J. Song, Y. Yue, and B. Dilkina, "A general large neighborhood search framework for solving integer linear programs," *arXiv preprint arXiv:2004.00422*, 2020.

[100] A. Hottung and K. Tierney, "Neural large neighborhood search for the capacitated vehicle routing problem," *arXiv preprint arXiv:1911.09539*, 2019.

[101] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," *arXiv preprint arXiv:1704.01665*, 2017.

[102] L. Bertacco, M. Fischetti, and A. Lodi, "A feasibility pump heuristic for general mixed-integer problems," *Discrete Optimization*, vol. 4, no. 1, pp. 63–76, 2007.

[103] C. Audet and W. Hare, *Derivative-Free and Blackbox Optimization*. Springer, 2017.

[104] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[105] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, G. Hendel, C. Hojny, T. Koch, P. Le Bodic, S. J. Maher, F. Matter, M. Miltenberger, E. Mühmer, B. Müller, M. E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, C. Tawfik, S. Vigerske, F. Wegscheider, D. Weninger, and J. Witzig, "The SCIP Optimization Suite 7.0," Zuse Institute Berlin, ZIB-Report 20-10, March 2020. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:0297-zib-78023

[106] E. Balas and A. Ho, "Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study," in *Combinatorial Optimization*. Springer, 1980, pp. 37–60.

[107] D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. Hooker, *Decision diagrams for optimization*. Springer, 2016, vol. 1.

[108] K. Leyton-Brown, M. Pearson, and Y. Shoham, "Towards a universal test suite for combinatorial auction algorithms," in *Proceedings of the 2nd ACM conference on Electronic commerce*, 2000, pp. 66–76.

[109] D. S. Hochbaum and A. Pathria, "Forest harvesting and minimum cuts: a new approach to handling spatial constraints," *Forest Science*, vol. 43, no. 4, pp. 544–554, 1997.

[110] M. Colombi, R. Mansini, and M. Savelsbergh, "The generalized independent set problem: Polyhedral analysis and solution approaches," *European Journal of Operational Research*, vol. 260, no. 1, pp. 41–55, 2017.

[111] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. Christophel, K. Jarck, T. Koch, and J. Linderoth, "Miplib 2017: data-driven compilation of the 6th mixed-integer programming library," *Mathematical Programming Computation*, pp. 1–48, 2021.

[112] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, and L. Antiga, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[113] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[114] S. Maher, M. Miltenberger, J. P. Pedroso, D. Rehfeldt, R. Schwarz, and F. Serrano, "PySCIPOpt: Mathematical programming in python with the SCIP optimization suite," in *Mathematical Software – ICMS 2016*. Springer International Publishing, 2016, pp. 301–307.

[115] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM computing surveys (CSUR)*, vol. 35, no. 3, pp. 268–308, 2003.

[116] E.-G. Talbi, "Machine learning into metaheuristics: A survey and taxonomy," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–32, 2021.

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

[118] C. Zhang, W. Song, Z. Cao, J. Zhang, P. S. Tan, and X. Chi, "Learning to dispatch for job shop scheduling via deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1621–1632, 2020.

[119] L. Gao, M. Chen, Q. Chen, G. Luo, N. Zhu, and Z. Liu, "Learn to design the heuristics for vehicle routing problem," *arXiv preprint arXiv:2002.08539*, 2020.

[120] V. Perreault, "Tactical wireless network design for challenging environments," Master's thesis, Ecole Polytechnique, Montreal (Canada), 2022.

[121] E. Danna, E. Rothberg, and C. Le Pape, "Exploring relaxation induced neighborhoods to improve mip solutions," *Mathematical Programming*, vol. 102, pp. 71–90, 2005.

[122] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.

[123] E. Khalil, C. Morris, and A. Lodi, "MIP-GNN: A data-driven framework for guiding combinatorial solvers," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 9, 2022, pp. 10 219–10 227.

[124] A. Prouvost, J. Dumouchelle, L. Scavuzzo, M. Gasse, D. Chételat, and A. Lodi, "Ecole: A gym-like library for machine learning in combinatorial optimization solvers," *arXiv preprint arXiv:2011.06069*, 2020.