Supervised Neighborhood Selection and MIP-based Primal Heuristics

by

Ravi Agrawal

B.Tech., Indian Institute of Technology Kharagpur, 2013

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Science

SIMON FRASER UNIVERSITY
Fall 2019

Approval

Name:	**Ravi Agrawal**
Degree:	**Master of Science (Computing Science)**
Title:	**Supervised Neighborhood Selection and MIP-based Primal Heuristics**
Examining Committee:	**Chair:** Qianping Gu Professor
	Ramesh Krishnamurti Senior Supervisor Professor
	Binay Bhattacharya Supervisor Professor
	Ehsan Iranmanesh Supervisor Research Scientist 1QB Information Technologies (1QBit)
	Abraham Punnen External Examiner Professor Department of Mathematics
Date Defended:	**December 16, 2019**

Abstract

Mixed integer programming provides a unifying framework for solving a medley of hard combinatorial optimization problems of practical interest. A mixed integer program (MIP) is typically solved using linear programming (LP) based branch-and-bound algorithm. Primal heuristics are a key component of MIP solvers and enable finding good feasible solutions early in the tree search. The literature is rich with a variety of hybrid primal heuristics that combine both heuristic and exact methods.

In this thesis, we propose a new supervised large neighborhood search heuristic for the general MIP, as well as, a new analytical MIP-based primal heuristic for the linear ordering problem. We present our work in two parts.

Part I: Supervised Neighborhood Selection for Mixed Integer Programs
Large neighborhood search (LNS) heuristics are employed as improvement procedures within the branch-and-bound algorithm. They formulate the neighborhoods as an auxiliary MIP with a restricted search space, which is then solved to search for an improving solution. The neighborhoods are typically defined by handcrafted rules, guided by human intuition and offline experimentation. Alternatively, a neighborhood should be defined such that it has a high likelihood of success. We apply a data-driven approach to predict an ideal neighborhood for the neighborhood search. We propose a supervised large neighborhood search heuristic for the general mixed integer programs and compare it with Relaxation Induced Neighborhood Search (RINS), a popular LNS heuristic. Our heuristic not only finds an improving solution more often but also improves the solver performance on key metrics.

Part II: MIP-based Primal Heuristic for the Linear Ordering Problem
Linear ordering problem (LOP) is a quintessential combinatorial optimization problem and has been well studied in the literature. We present a new MIP-based primal heuristic for the LOP. The heuristic decomposes the LOP instance into sub-problems albeit sub-optimal ones, solves each sub-problem optimally and concatenates the partial solutions to construct a solution to the original problem instance. We present empirical results that show that our heuristics achieves good performance on benchmark instances.

Keywords: Mixed Integer Programming; Primal Heuristics; Large Neighborhood Search; Machine Learning; Supervised Learning; Linear Ordering Problem

To my parents, for their unbounded love and support.

Acknowledgements

First and foremost, I express my utmost gratitude to my supervisor Dr. Ramesh Krishnamurti. I thank him for believing in me and for allowing me to work with him. He has been a great mentor, and not only did I enjoy learning from him about algorithms and research but also countless invaluable life lessons. Ramesh has always been open, encouraging, kind, patient, and humble, and I would never have been able to finish this thesis without his guidance.

My sincere appreciation to Dr. Binay Bhattacharya for motivating me to pursue good research. His unmatched enthusiasm has been inspiring and continually encouraged me to push myself. I am grateful to Dr. Ehsan Iranmanesh for taking time out of his busy schedule and always making it to our meetings. He has been a guiding voice throughout the work in this thesis, and I appreciate his help in giving shape to my research ideas. I thank Dr. Abraham Punnen for making time to read my thesis and for his constructive comments.

I am grateful to my fellow lab-mates and friends, Meghna, Hamid, Akbar, and Amirhossein, and I thank them for being there for support in research or otherwise.

A heartfelt thanks to my family and friends. My parents, for encouraging me in all my endeavors and inspiring me to follow my dreams. I am forever indebted to them as I would not be here if it weren't for their sacrifices. My brother, for being supportive and one of the most understanding and sensible persons in my life. I wish him the best for his future. Last but not least, to my friends who have always been there, listened to me, given me advice, and helped me stay on track. Especially to Rimika, for enduring through my thesis and for her helpful comments.

I would also like to acknowledge that this research was enabled in part by support provided by WestGrid (www.westgrid.ca) and Compute Canada (www.computecanada.ca).

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A large number of combinatorial optimization (CO) problems are difficult-to-solve (NP-hard) yet ubiquitous. They frequently arise in a diversity of domains, including but not limited to operations research, economics, finance, energy, electronics, and bioinformatics. Their computational intractability, therefore, does not curtail their significance in real-world applications, and we must put our best efforts to solve them as well as possible. This has motivated the research in approximation algorithms [80], heuristic methods [21], and integer programming [81].

Approximation algorithms have the advantage of theoretical guarantees as they provide a strict bound on the solution quality. However, algorithms with satisfactory bounds may not exist for a given problem. Besides, some optimization problems are known to be inapproximable, a classic example being the general case of the *traveling salesman problem*. They additionally may not be practical for solving very large scale problems. Heuristic methods, in contrast, make a compromise with theoretical guarantees in favour of being highly efficient and scalable. Moreover, unlike approximation algorithms, many available general-purpose heuristics readily apply to a variety of different problems. Therefore, heuristic methods are particularly prevalent in practice. Integer programming is another method widely used in practical applications not only for its high scalability but also for its capability to provide a guarantee on solution quality. Although solving an integer programming problem can theoretically take exponential time, exact integer programming solvers exploit an array of tools to achieve satisfactory performance in practice [3]. More recently, the research community has shown a keen interest in applying machine learning techniques to combinatorial optimization. The proposed methods intend to either replace exact solvers and heuristic methods in a particular problem domain or support general-purpose exact solvers [14].

Classical combinatorial problems include the *satisfiability problem*, *knapsack problem*, *traveling salesman problem*, *set cover*, *set packing*, and the *0-1 integer program*. The equivalence among these problems is well known [52]. Even though it may be possible to apply known approximation and heuristic algorithms for one problem to another, not all such reductions may be efficient. Mixed integer programming provides a unifying framework

to model these combinatorial problems. Furthermore, it can be readily extended to many other diverse optimization problems. A mixed integer programming problem (MIP) is characterized by a linear objective function over certain variables, subject to a set of linear constraints. A MIP is typically solved using a linear programming problem (LP) based branch-and-bound algorithm. The algorithm solves a series of LP-relaxations, branching on any fractional *integer variables*, in search of a feasible solution. As noted earlier, the solving time using the branch-and-bound algorithm may still be disproportionately high. To bring it down significantly, a modern MIP solver implements many auxiliary *solving components* [19], such as branching strategies, node selection strategies, presolving, cutting plane separation, and primal heuristics.

Thanks to decades of research and development, modern MIP solvers have become remarkably effective and robust tools integral to today's global economy [20, 5]. Problems such as *vehicle routing*, *scheduling*, *production planning*, *network design*, *crew pairing*, and *task allocation* are quite commonly solved using MIP solvers [72]. Although we have come a long way, the diversity and the scale of the problems we need to solve today is larger than ever. To this end, further advances in mixed integer programming still have the potential to have a far-reaching impact.

Primal heuristics are integral in improving the performance of MIP solvers as they enable early exploration of *good* feasible solutions. In this thesis, our central focus is on hybrid primal heuristics that combine approximate and exact methods. Solving a MIP to obtain good-quality feasible solutions is known to require computing time that is comparable to solving its LP-relaxation [34]. Large neighborhood search heuristics and MIP-based heuristics leverage exact MIP solvers as a *black-box* to solve parts of the problem optimally or near-optimally. In our work, we explore the idea of designing a dynamic, data-driven, and context-dependent large neighborhood search heuristic for the general mixed integer programming problem. We also propose an analytical MIP-based heuristic for the linear ordering problem.

1.1 Role of primal heuristics

Mixed integer programming has proved extremely powerful and reliable for solving a diverse set of CO problems. However, obtaining exact solutions to many hard problems remains challenging. Primal heuristics are crucial in finding and improving feasible solutions early in the branch-and-bound search. A reasonable assumption is that good primal bounds attained early help prune sub-optimal branches more quickly, thereby improving the overall performance of the algorithm. However, experiments have shown that primal heuristics have a rather limited contribution in solving a problem instance to optimality [2]. In practical applications, a user may not always prioritize finding a provably optimal solution. A feasible solution with a reasonable quality might be sufficient. Primal heuristics are critical in this

regard. Berthold [17] says the impact of heuristics on solving a MIP is not rightly measured by time to optimality or number of branch-and-bound nodes. Instead they propose *primal integral* as a performance metric, and show that the heuristics improve the primal performance by up to 80%.

Primal heuristics have been a focus of extensive research, and the literature is rich in heuristics for the general MIP. Early heuristics were mainly targeted towards achieving feasibility quickly. One of the first well-known heuristics is *pivot and complement* [10], originally designed for the 0-1 integer program and later generalized to mixed integer program as *pivot and shift* [11]. It builds upon the observation that an LP-feasible solution is also integer feasible if all its integer variables are non-basic. The heuristic works in two phases. In the first phase, it performs non-standard pivot operations in an attempt to drive non-basic slack variables into the basis at a minimal cost. Finally, it attempts to improve the obtained feasible solution using local search by complementing (or shifting) certain sets of variables. Other popular primal heuristics for mixed integer programming can be broadly classified as *rounding heuristics*, *diving heuristics*, *objective diving heuristics*, and *improvement heuristics* [47].

Rounding heuristics, as the name suggests, perform a series of rounding operations on fractional *integer variables* in the LP-relaxation. A simple rounding heuristic is one where we attempt to round every fractional *integer variable* while maintaining the LP-feasibility. Other more sophisticated rounding heuristics may perform rounding operations that occasionally result in an LP-infeasible solution, and recover from the infeasibility by further rounding operations. In contrast, Relaxation Enforced Neighborhood Search (RENS) [18] defines a neighborhood around all feasible roundings of the LP-relaxation solution by adjusting the bounds on all *integer variables*. Diving heuristics utilize bound adjustments slightly differently. They perform a sequence of bound changes to simulate a *dive* or *plunge* in the branch-and-bound tree without actually generating the intermediate nodes. Any infeasibilities resulting from a bound change are resolved using the *dual simplex algorithm*. The dive may eventually lead to a feasible solution or is infeasible, in which case the heuristic is said to have failed. Alternatively, objective diving heuristics perform what is called *soft roundings*, to always maintain feasibility. Instead of explicitly adjusting the variable bounds, they modify the objective function coefficient of the variable to drive it towards its lower or upper bound.

While the main focus of rounding and diving heuristics is on achieving feasibility, a wide selection of primal heuristics are classified as Large Neighborhood Search (LNS) heuristics targeted towards improving a known feasible solution [26, 32, 35, 40, 73]. LNS heuristics restrict the search space of the input MIP instance to a neighborhood of interest and partially solve an *auxiliary problem* (also a MIP instance) using the branch-and-bound algorithm [47]. The neighborhoods may be defined in several ways i.e. by *fixing variables*, *enforcing additional constraints*, *modifying objective function coefficients*, or any combination of these.

This technique is also utilized by *feasibility pump* [31], a heuristic designed to achieve feasibility quickly. Popular metaheuristic methods such as *tabu search*, *variable neighborhood search* and *genetic algorithms* have also been applied to the general integer programming problems [9, 44, 66].

For an extensive survey on primal heuristics for general mixed integer programs, we direct the interested reader to these resources [16, 33].

1.2 Motivation for learning in combinatorial optimization

The idea of utilizing machine learning for combinatorial optimization problems is not a new one [48]. However, the dramatic explosion in machine learning research in the last few years has renewed interest in learning-based algorithms for NP-hard problems. The primary motivation, as described in a recent survey by Bengio et al. [14], is to replace some heavy computation with a faster learning-based approximation or to efficiently explore the algorithmic design space in search of better performing policies. We can classify the recent work in this area into two broad categories. First, are approaches that provide an alternative to existing solving methods such as heuristics or exact solvers [25, 60]. Second, are approaches designed to support exact solvers by augmenting them [45, 53, 54].

In practice, the field of combinatorial optimization has long been dominated by heuristic methods, as *good* solutions are often acceptable. Moreover, exact methods may not scale well or be fast enough to support decision making in many real-world applications. These heuristics are essentially handcrafted rules, designed by human intuition, and refined by extensive experimentation and tuning by experts with domain knowledge. Alternatively, we can automate this process in an attempt to derive heuristic rules directly from large datasets. Li et al. [60] identify that machine learning can play a key role in detecting useful patterns, by leveraging the regularities in real-world data. Such patterns may elude human algorithm designers or may be hard to describe manually. They give the example of graph motifs that can indicate a set of vertices that belong to the optimal solution.

Many recent papers demonstrate the utility of machine learning for specific combinatorial problems such as the traveling salesman problem (TSP) [13, 79], knapsack problem [13], vehicle routing problem (VRP) [56], satisfiability problem [75], and the graph coloring problem [59]. Problems such as TSP and VRP are frequently solved by companies in the transportation and logistics industries, often with minor tweaks to the problem parameters. In such scenarios, a faster method to solve these problem may be more convenient. Other studies try to generalize learning-based models to many different problems at once. Dai et al. [25] propose learning greedy heuristics using reinforcement learning, more specifically deep Q-learning, for graph problems such as minimum vertex cover, maximum cut, and the TSP. On the other hand, Li et al. [60] propose a supervised graph convolutional network and tree search based algorithm for graph problems such as maximum independent set, minimum

vertex cover, maximal clique and satisfiability. Their approach uses a graph convolutional network to estimate the likelihood for each vertex to be a part of the optimal solution. Further, they use tree search to disambiguate solutions where multiple different optimal solutions may exist. The advantage of their approach over the reinforcement learning based approach of Dai et al. [25] is that the same model can generalize to multiple problems at once and need to not be trained separately for each problem.

Alternatively, owing to the widespread applicability of mixed integer programming to an even broader class of CO problems, methods have been proposed to assist exact MIP solvers. By far, the most common problem targeted in the context of MIP solvers is one of *branching variable selection* [8, 12, 38, 53]. Selecting the right branching variables can have a significant impact on the size of the branch-and-bound tree, thus reducing the time to optimality. Strong branching is a technique experimentally shown to produce smaller search trees, but is computationally expensive [4]. The proposed learning-based approaches aim to learn a fast approximation for strong branching in an attempt to make it a viable branching strategy.

1.3 Structure and overview of results

In the following chapters, we will introduce our research on primal heuristics, specifically, hybrid heuristics that combine both heuristic and exact methods. In Chapter 2, we discuss the relevant literature and prior work that intersects with the ideas in this thesis. We mainly outline work related to large neighborhood search heuristics, and the implementation of machine learning techniques in the context of mixed integer programming. Chapter 3 provides the relevant background and definitions key to understanding our work. In Chapter 4, we introduce our supervised learning framework for neighborhood selection and devise a large neighborhood search heuristic for the general mixed integer programming problem. In Chapter 5, we present a more analytical approach to MIP-based heuristics in the context of the linear ordering problem. Finally, we discuss the advantages, limitations, and potential directions for future work in Chapter 6, and provide a short conclusion in Chapter 7.

1.4 Bibliographic notes

Chapter 5 is based on joint work with Ehsan Iranmanesh and Ramesh Krishnamurti, accepted as a full paper at the International Conference on Operations Research and Enterprise Systems (ICORES) 2019 [6].

Chapter 2

Related Work

Our work intersects two diverse areas of computer science: combinatorial optimization, and machine learning. In this chapter, we first review the prior literature on large neighborhood search heuristics for the mixed integer programming problem. Next, we discuss the more recent work focused on the integration of machine learning with MIP solvers.

2.1 Large neighborhood search heuristics

Neighborhood search, also known as local search, is a metaheuristic strategy commonly employed as an *improvement heuristic* within many heuristic algorithms. The idea behind neighborhood search is simple, namely: iteratively try to improve a known feasible solution by exploring the solutions around it. In any neighborhood search heuristic, the choice of the neighborhood dictates its performance. Typically, the larger the neighborhood, the higher the likelihood of obtaining good-quality solutions. However, searching large neighborhoods can be computationally expensive and therefore it is a trade-off. In order to take advantage of large neighborhoods, one must develop efficient algorithms to explore them. In a survey on *very large scale neighborhood search*, Ahuja et al. [7] consider three broad techniques for efficiently exploring large neighborhoods. They study variable-depth methods that partially search exponentially large neighborhoods, network flow based techniques to identify improving neighborhoods, and finally, neighborhoods for NP-hard problems induced by subclasses or restrictions of the problems that are solvable in polynomial time.

Variable-depth methods are techniques that search the k-OPT neighborhood partially as effort required for an exhaustive search may become prohibitive as k grows large. Typically, k-OPT neighborhoods with only a small value of k i.e. 1 or 2 can be searched efficiently but are known to yield a better local optima for larger values of k. Variable-depth methods are designed to find solutions close to the global optimum although they do not guarantee a local optima. A popular example of this approach is the Lin-Kernighan heuristic for the TSP [61]. The network flow based techniques for neighborhood search include minimum cost cycle finding methods, shortest path or dynamic programming based methods, and methods based

on finding minimum cost assignments or matching. Finally, there are methods that utilize special cases of NP-hard problems to search exponential sized neighborhoods in polynomial time. An example is the TSP restricted to *Halin graphs* that may have exponential number of TSP tours but is solvable in linear time [24].

Large neighborhood search (LNS) heuristics in the context of mixed integer programming define the *large neighborhood* by formulating an *auxiliary problem*, typically by restricting the search space of the original MIP. The search is then performed by solving the auxiliary problem, also called the sub-MIP, using the branch-and-bound algorithm. Any improving solutions found during the search are applied back to the original MIP. To avoid an explosion in computation time, solving the auxiliary problem is subject to reasonable limits with respect to the number of nodes or number of LPs solved. Modern MIP solvers, armed with an arsenal of tools such as presolving and cutting plane separation, are extremely efficient in exploring these restricted search spaces. As a result, LNS heuristics have proved quite effective in improving the primal performance of MIP solvers.

The literature is rich with a wide selection of large neighborhood search heuristics. One of the first heuristics, called *local branching* [32], was designed primarily for the 0-1 integer programming problem. Local branching defines a neighborhood by adding an *invalid global cut*, called the *local branching constraint*. The constraint puts a bound on the *Manhattan distance* from the best known feasible solution, also called the *incumbent solution*. It restricts the number of variables that can change their values as compared to the incumbent solution, essentially, defining a k-OPT neighborhood around the incumbent solution. This strategy is referred to as *soft fixing* as compared to *hard fixing* where a set of variables are fixed explicitly. Relaxation Induced Neighborhood Search (RINS) [26] adopts the hard-fixing strategy. It defines the neighborhood by fixing variables that assume the same value in the LP-relaxation solution and the incumbent solution. Experiments show that RINS is more effective compared to local branching. The sub-MIP for RINS is typically less computationally expensive due to the reduced number of variables. Moreover, it does not contain a dense constraint like in local branching. RINS is also able to produce better quality solutions. Unlike local branching, it does not get stuck in a neighborhood of a poor-quality solution. RINS is especially useful in case of highly constrained problems, where one must change the solution significantly to find good solutions.

Rothberg [73] proposed a solution-polishing heuristic that uses the *variable fixing neighborhood* similar to RINS but combines it with the evolutionary algorithm approach. Unlike RINS, this heuristic uses crossover and mutation to combine multiple solutions selected from a pool of available feasible solutions. The crossover procedure fixes variables that assume the same value across all the selected solutions. Mutation is introduced by fixing additional variables randomly. The heuristic is mainly designed to further polish already good solutions. It is able to find improving solutions in very late stages of the solving process when other heuristics are unable to do so. Restrict and relax [43] uses a combination of fixing

variables, and unfixing or relaxing previously fixed variables. The heuristic always works with a restricted but dynamically changing set of variables. As a result, it can be used to solve very large problems that do not fit in the memory. Relaxation Enforced Neighborhood Search (RENS) [18], takes a different approach. Instead of fixing variables, it restricts the search space to all feasible roundings of the LP-relaxation solution. The Distance Induced Neighborhood Search (DINS) [40] combines elements from RINS, local branching, and bound changes.

An alternate way to define a neighborhood is by using an *auxiliary objective function*, an approach used by proximity search [35]. Similar to local branching, proximity search heuristic was proposed for the 0-1 integer programs. It introduces the idea of formulating the auxiliary problem using an objective function different from the original. It replaces the original objective function with one that minimizes the *Hamming distance* from the incumbent solution. Proximity search adds an explicit *cutoff constraint* to prevent it from searching the non-improving neighborhood. An important requirement for the LNS heuristics discussed so far is a known feasible solution. However, a feasible solution may not always be available and some heuristics are specifically designed to tackle the problem of obtaining a feasible solution. Feasiblity pump [31] tries to achieve feasibility by recursively minimizing the distance between the LP-relaxation solution and its rounded solution. The feasibility pump was later generalized to the general mixed integer programs [15]. Although the feasiblity pump heuristic works well in obtaining a feasible solution, the solutions obtained are often poor-quality because the original objective function is disregarded. Achterberg et al. [3] proposed a slight modification, called *objective feasibility pump*, and included the original objective function along with the feasibility pump objective, to enable finding better quality solutions.

A common characteristic among the different LNS heuristics discussed here, is the way they define the neighborhood, and use a MIP solver as a *black-box* to solve a sub-problem. Although these heuristics have proved extremely useful, they often take much longer compared to their *rounding* or *diving* counterparts. For this reason, they are not executed as frequently in the branch-and-bound search. Moreover, it is also unclear which neighborhoods might be lucrative for which problems and when they should be executed in the branch-and-bound search. Hendel [47] proposed Adaptive Large Neighborhood Search (ALNS), an orchestration technique for the many LNS heuristics implemented in SCIP [42] by formulating it as a *multi-armed bandit problem*. The adaptive heuristic tries to devise a selection policy that maximizes the reward, based entirely on the reward feedback it gets after having selected a neighborhood. The handcrafted reward function combines rewards for the presence of a solution, relative amount of gap closed, and the time spent on solving the sub-MIP.

A key observation is that the proposed heuristics rely on static rules and handcrafted strategies to select a neighborhood. We take away from this wide selection of heuristics,

the more general and broader ideas, like *fixing variables*, *enforcing additional constraints*, and *modifying objective function coefficients*, and alternatively propose the idea of defining a neighborhood such that it has a high likelihood for success. In this thesis, we propose the idea of a supervised *variable fixing neighborhood*. Our *supervised neighborhood selection* framework defines the neighborhood based on the likelihood of a variable to remain fixed at its value in the incumbent solution. In chapter 4, we develop this idea further and describe our supervised large neighborhood search heuristic.

2.2 Machine learning in mixed integer programming

LP-based branch-and-bound algorithm, by itself, is often not enough to achieve a satisfactory performance on real-world problem instances. Therefore, a MIP solver must employ a number of auxiliary *solving components*, including but not limited to, sophisticated branching strategies, node selection strategies, presolving, cutting plane separation, and primal heuristics [19]. The right selection of these strategies, for an instance at hand, can dramatically improve the solver's performance [3]. Unfortunately, these algorithmic problems within the context of mixed integer programming are often tackled using heuristics due to their ill-defined nature [62]. The heuristics incorporated are static, instance-agnostic, and handcrafted rules, tuned by offline experimentation. Machine learning (ML) based methods, on the other hand, have the advantage of automatically inferring dynamic, context-dependent strategies from a large dataset. ML can help in exploring the algorithmic design space more efficiently and learn complex rules difficult to specify by hand. It can also help in learning a fast-approximation for a problem that otherwise takes a significant amount of time to solve. This is the primary motivation behind the integration of machine learning with MIP solvers.

In the context of MIP, the most common problem tackled using machine learning is one of branching variable selection [8, 12, 38, 53]. Branching strategies play a major role in solving an instance to optimality. Selecting the right variables to branch on, can significantly improve the solving time [3]. Unfortunately, there is a limited mathematical understanding regarding choosing the right branching variables. Achterberg et al. [4] reviewed the different branching strategies and compared them via a computational study. Early branching strategies include simple heuristics, like *minimum infeasibility branching* and *maximum infeasibility branching*, that select a branching variable based on its distance from integer values. More sophisticated information-based methods, like *pseudocost branching*, track the bound improved by a variable each time it is used for branching. Pseudocost branching works well in practice, however, search trees are still quite large compared to *strong branching*, a strategy experimentally shown to generate smaller search trees [4]. However, strong branching is impractical due to the computational effort involved. They propose a hybrid method called *reliability branching* that uses strong branching only on variables with an

unreliable pseudocost and falls back to pseudocost branching otherwise. MIP solvers also often implement their own strategies or the *default strategy* that is a combination of these methods. However, these strategies are not open source, and we have no clear idea of the parameters that are used to define them. While the previous methods rely on the dual bound of child nodes, Karzan et al. [55] propose collecting information on branching variables that result in fathomed child nodes for a partial exploration of the branch-and-bound tree and then restart the solving process.

One of the first ML-based approaches for branching strategies was Dynamic Approach for Switching Heuristics (DASH) proposed by Di Liberto et al. [28]. They collect 40 features on a diverse set of problem instances and evaluate the different branching strategies on each instance. Then for each strategy, they identify clusters of instances for which the strategy works the best and compute their regions. Thereafter, for any node in the branch-and-bound tree, they use the strategy corresponding to the region in which the node MIP instance lies. More recent work around learning-based branching rules is mainly focused around finding a fast approximation for *strong-branching*. These methods record the data i.e. the state-action pairs by using the *strong-branching* expert rule and learn to predict a branching variable. The approaches differ mainly on how they define the problem, the machine learning model they use, and how they train the model. Alvarez et al. [8] use *Extremely Randomized Trees*, also known as ExtraTrees, to approximate the *strong-branching score* for each variable. Khalil et al. [53] use SVM-rank to learn a ranking function for the variables instead. Balcan et al. [12] learn a convex combination of the different scoring rules. Gasse et al. [38] propose an offline-learning model unlike the previous work to predict a branching variable. They use the natural constraint-variable bipartite representation of the MIP problem and train a graph convolutional network using standard classification algorithm. For further reading on learning and branching, please refer to the recent surveys by Lodi and Zarpellon [65], and Dilkina et al. [29].

Other methods target the problem of *node selection*, within the branch-and-bound search algorithms in MIP solvers. He et al. [45] use an imitation learning based approach to learn an adaptive search policy within the branch-and-bound algorithms. Song et al. [76] build upon their work and propose a paradigm called *retrospective imitation* where a policy learns from its own mistakes without repeated expert feedback, which can be expensive to obtain. Tang et al. [77] investigate the problem of solving MIP instances using the cutting plane method. They use a reinforcement learning approach to learn a policy for intelligent adaptive selection of cutting planes. For challenges around cutting plane selection, please refer to a recent survey by Dey and Molinaro [27]. Feature based algorithm selection by Georges et al. [39], defines the problem of selecting appropriate parameters for the MIP solver given a problem instance. The idea behind this work is to predict what algorithm is likely to perform the best for a problem instance. Here, an algorithm is defined by a unique setting of the

solver's parameters. Machine learning has also been applied to the problem of predicting when it is most profitable to use decomposition [57].

Finally, primal heuristics within MIP solvers, is another problem of interest. Khalil et al. [54] study the problem of predicting when to run a heuristic. They use eight diving heuristics implemented within SCIP [42] and train a classifier to predict if a heuristic would be successful if it is run at a particular node. They define a run when successful (RWS) policy to run the heuristic whenever the predictor returns true for a particular heuristic. The approach is promising as it helps reduce the amount of time spent on heuristics. Our approach builds upon this idea and applies it to the problem of *neighborhood selection* in the context of large neighborhood search heuristics. However, in the case of search neighborhoods, we can design a more fine-grained policy. Our approach is most similar to two very recent papers on solution prediction [63, 30]. Where Lodi et al. [63], tackle the problem of predicting solutions under parameter perturbation in case of *facility location problem*, Ding et al. [30] propose a more generalized approach to solution prediction. We target the problem rather differently in the context of large neighborhood heuristics, and formalize the problem as *neighborhood selection*. We highlight the key differences between our work and those in the literature in Chapter 4.

Chapter 3

Background

3.1 A brief overview of mixed integer programming

We start with the formal definition of a *mixed integer program* (MIP). As introduced earlier, a MIP is characterized by a linear objective function to be optimized subject to a set of linear constraints. Let $m, n \in \mathbb{N}$ be the number of constraints and the number of variables, respectively, and $\boldsymbol{N} := \{1, 2, \ldots, n\}$ be the set of variable indices.

Definition 3.1. Mixed Integer Program
Given the objective function coefficients $\boldsymbol{c} \in \mathbb{R}^n$, the constraint matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, the constraints' RHS $\boldsymbol{b} \in \mathbb{R}^m$, the variable bounds $\boldsymbol{l}, \boldsymbol{u} \in \mathbb{R}^n$ and the set of indices for integer variables $\boldsymbol{I} \subseteq \boldsymbol{N}$, we define the mixed integer program $\mathcal{P}$ as:

$$\text{Minimize} \quad \boldsymbol{c}^\intercal \boldsymbol{x} \tag{3.1}$$

$$\text{s.t.}$$

$$\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \tag{3.2}$$

$$\boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u} \tag{3.3}$$

$$\boldsymbol{x} \in \mathbb{R}^n \tag{3.4}$$

$$x_j \in \mathbb{Z} \qquad \forall \quad j \in \boldsymbol{I} \tag{3.5}$$

We refer to $\boldsymbol{c}^\intercal \boldsymbol{x}$ in equation (3.1) as the *objective function*, $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$ in equation (3.2) as the *linear constraints*, $\boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}$ in equation (3.3) as the *bounding constraints*, and equation (3.5) as the *integrality constraints* of the MIP. For any variable x_j for $j \in \boldsymbol{N}$, c_j denotes its *objective function coefficient*, and l_j and u_j denote its *lower bound* and *upper bound*, respectively.

The variables participating in the MIP can be classified as *integer*, *binary*, *general integer*, or *continuous* based on the integrality constraints, and their lower and upper bounds.

Definition 3.2. Let $\boldsymbol{B} := \{j \in \boldsymbol{I} \mid l_j = 0 \text{ and } u_j = 1\}$. A variable x_j in the MIP is:

$$\begin{aligned} \text{an Integer Variable} \quad & \textit{if} \quad j \in \boldsymbol{I} \\ \text{a Binary Variable} \quad & \textit{if} \quad j \in \boldsymbol{B} \\ \text{a General Integer Variable} \quad & \textit{if} \quad j \in \boldsymbol{I} \setminus \boldsymbol{B} \\ \text{a Continuous Variable} \quad & \textit{if} \quad j \in \boldsymbol{N} \setminus \boldsymbol{I} \end{aligned}$$

The MIP as defined in (3.1 - 3.5) is in its most general form. However, we can classify a MIP as *a Linear Program*, *an Integer Program*, *a Binary Program*, *a Mixed Binary Program* or *a Mixed Integer Program* based on the number of variables of different types.

Definition 3.3. A MIP as defined in (3.1 - 3.5) is referred to as:

$$\begin{aligned} \text{a Linear Program (LP)} \quad & \textit{if} \quad \boldsymbol{I} = \emptyset \\ \text{an Integer Program (IP)} \quad & \textit{if} \quad \boldsymbol{I} = \boldsymbol{N} \\ \text{a Binary Program (BP) or 0-1 IP} \quad & \textit{if} \quad \boldsymbol{B} = \boldsymbol{N} \\ \text{a Mixed Binary Program (MBP)} \quad & \textit{if} \quad \boldsymbol{B} = \boldsymbol{I} \neq \emptyset \quad \text{and} \quad \boldsymbol{N} \setminus \boldsymbol{I} \neq \emptyset \\ \text{a Mixed Integer Program (MIP)} \quad & \textit{if} \quad \boldsymbol{I} \neq \emptyset \quad \text{and} \quad \boldsymbol{N} \setminus \boldsymbol{I} \neq \emptyset \end{aligned}$$

Definition 3.4. LP-relaxation
The LP we get by eliminating the *integrality constraints* as defined in (3.5) from a MIP is called its LP-relaxation

Any solution to the MIP can be classified as *LP-feasible*, *Integer-feasible*, *IP-feasible* or *Optimal* based on the constraints (3.2 - 3.5) it satisfies and the value of the *objective function*. The terms *LP-infeasible*, *Integer-infeasible*, *IP-infeasible* are defined analogously.

Definition 3.5. A solution $\hat{\boldsymbol{x}} \in \mathbb{R}^n$ for the MIP is called:

$$\begin{aligned} \text{LP-feasible} \quad & \textit{if} \quad \boldsymbol{A}\hat{\boldsymbol{x}} \geq \boldsymbol{b} \quad \text{and} \quad \boldsymbol{l} \leq \hat{\boldsymbol{x}} \leq \boldsymbol{u} \\ \text{Integer-feasible} \quad & \textit{if} \quad \hat{x}_j \in \mathbb{Z} \quad \forall \quad j \in \boldsymbol{I} \\ \text{IP-feasible} \quad & \textit{if} \quad \hat{\boldsymbol{x}} \text{ is both LP-feasible and Integer-feasible} \\ \text{Optimal} \quad & \textit{if} \quad \hat{\boldsymbol{x}} \text{ is IP-feasible and } \boldsymbol{c}^\intercal \hat{\boldsymbol{x}} \leq \boldsymbol{c}^\intercal \boldsymbol{x} \\ & \qquad \text{for all feasible solutions } \boldsymbol{x} \end{aligned}$$

3.1.1 The branch-and-bound algorithm

The branch-and-bound algorithm to solve a MIP, adopts a divide-and-conquer type technique that manifests into a search tree as shown in Figure 3.1. Every node in the search tree corresponds to a MIP, and we solve its LP-relaxation. If any integer variable x_j for

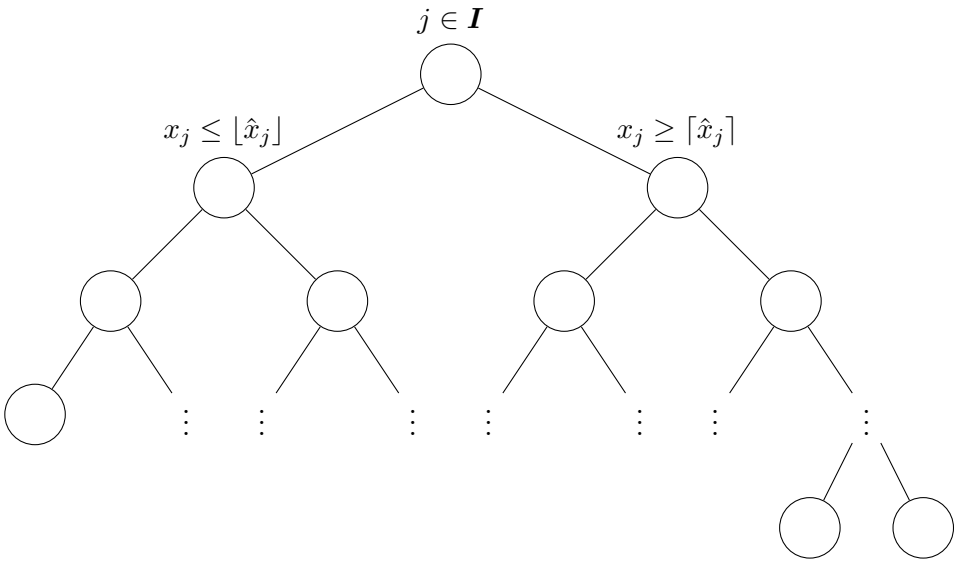


Figure 3.1: Illustration of the branch-and-bound algorithm

$j \in \boldsymbol{I}$ assumes a fractional value in the LP-relaxation solution, we branch on the variable by restricting its bounds. Essentially, we create two child nodes or sub-MIPs by bounding the variable to $\leq \lfloor \hat{x}_j \rfloor$ and $\geq \lceil \hat{x}_j \rceil$. Note that the branching leads to two disjoint feasible regions, none containing the previous LP-relaxation solution. However, if the LP-relaxation is infeasible or IP-feasible, the node does not need to be expanded. If the LP-relaxation is infeasible, then by definition it is also IP-infeasible and the node can be fathomed. Otherwise, we have found an IP-feasible solution. Note that as we relax the integrality constraints, the objective value of any IP-feasible solutions resulting from a sub-tree is worse or equal to the objective value of the LP-relaxation at its root node. Therefore, any nodes with an LP-relaxation worse than the best known IP-feasible solution, also called the incumbent, can be fathomed. After we have solved or fathomed every node in the tree, we have solved the original MIP to optimality. Now, we define key terms associated to the branch-and-bound algorithm.

Definition 3.6. Branch and Bound Search Tree
For a given MIP $\mathcal{P}$, the branch-and-bound search tree $\mathcal{T}$ is the tree, as shown in Figure 3.1, resulting from the branch-and-bound algorithm. Each node v in $\mathcal{T}$ is a MIP, with altered variable bounds. The node corresponding to $\mathcal{P}$ is referred to as the root node.

Definition 3.7. Incumbent Solution
At any time t in the branch-and-bound search, the incumbent solution is the best IP-feasible solution found until time t. Let $\mathcal{J}_t$ be the set of IP-feasible solutions found until time t. An incumbent solution $\tilde{\boldsymbol{x}} \in \mathcal{J}_t$ is defined as:

$$\tilde{\boldsymbol{x}} = \hat{\boldsymbol{x}} \quad \textit{where} \quad \boldsymbol{c}^\intercal \hat{\boldsymbol{x}} \leq \boldsymbol{c}^\intercal \boldsymbol{x} \quad \forall \quad \boldsymbol{x} \in \mathcal{J}_t \tag{3.6}$$

We refer to the set of incumbent solutions or strictly improving IP-feasible solutions found in the branch-and-bound search as $\mathcal{I}$.

Definition 3.8. Dual Bound
The dual bound for a node v in the tree $\mathcal{T}$ is the objective value of its LP-relaxation solution. The dual bound for the tree $\mathcal{T}$ is given by the best LP-relaxation solution among all the nodes in the tree.

Definition 3.9. Primal Bound
Let $\tilde{\boldsymbol{x}}$ be the incumbent solution for $\mathcal{P}$. The primal bound for the branch-and-bound search for $\mathcal{P}$ is then given by its objective value $\boldsymbol{c}^\intercal \tilde{\boldsymbol{x}}$.

Definition 3.10. Primal Gap
Let $\hat{\boldsymbol{x}}$ be a feasible solution of a given MIP, and $\boldsymbol{x}^*$ its optimal solution. We define the primal gap γ_p as:

$$\gamma_p(\hat{\boldsymbol{x}}) = \begin{cases} \frac{|\boldsymbol{c}^\intercal \hat{\boldsymbol{x}} - \boldsymbol{c}^\intercal \boldsymbol{x}^*|}{\max\{|\boldsymbol{c}^\intercal \hat{\boldsymbol{x}}|, |\boldsymbol{c}^\intercal \boldsymbol{x}^*|\}} & \textit{if} \quad |\boldsymbol{c}^\intercal \boldsymbol{x}^*| > 0 \quad \textit{and} \quad \boldsymbol{c}^\intercal \hat{\boldsymbol{x}} \cdot \boldsymbol{c}^\intercal \boldsymbol{x}^* > 0 \\ 0 & \textit{if} \quad |\boldsymbol{c}^\intercal \hat{\boldsymbol{x}}| = |\boldsymbol{c}^\intercal \boldsymbol{x}^*| = 0 \\ \infty & \textit{otherwise} \end{cases} \tag{3.7}$$

Definition 3.11. Primal Dual Gap
Let $\hat{\boldsymbol{x}}$ be a feasible solution of a given MIP, and $\boldsymbol{c}^\intercal \hat{\boldsymbol{y}}$ the dual bound during the branch-and-bound. We define the primal-dual gap γ_{pd} as:

$$\gamma_{pd}(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) = \begin{cases} \frac{|\boldsymbol{c}^\intercal \hat{\boldsymbol{x}} - \boldsymbol{c}^\intercal \hat{\boldsymbol{y}}|}{\max\{|\boldsymbol{c}^\intercal \hat{\boldsymbol{x}}|, |\boldsymbol{c}^\intercal \hat{\boldsymbol{y}}|\}} & \textit{if} \quad |\boldsymbol{c}^\intercal \hat{\boldsymbol{y}}| > 0 \quad \textit{and} \quad \boldsymbol{c}^\intercal \hat{\boldsymbol{x}} \cdot \boldsymbol{c}^\intercal \hat{\boldsymbol{y}} > 0 \\ 0 & \textit{if} \quad |\boldsymbol{c}^\intercal \hat{\boldsymbol{x}}| = |\boldsymbol{c}^\intercal \hat{\boldsymbol{y}}| = 0 \\ \infty & \textit{otherwise} \end{cases} \tag{3.8}$$

Definition 3.12. Primal Gap Function [17]
Let $t_{max} \in \mathbb{R}_{\geq 0}$ be a limit on the solution time of a MIP solver. Its primal gap function $p : [0, t_{max}] \mapsto [0, 1]$ is defined as:

$$p(t) = \begin{cases} 1 & \textit{if no incumbent until point } t, \\ \gamma_p(\tilde{\boldsymbol{x}}(t)) & \textit{otherwise, where } \tilde{\boldsymbol{x}}(t) \textit{ is the incumbent solution at time } t. \end{cases} \tag{3.9}$$

Definition 3.13. Primal Integral [17]
Let $T \in [0, t_{max}]$ and let $t_i \in [0, T]$ for $i \in \{1, \ldots, |\mathcal{I}| - 1\}$ be the points in time a new incumbent solution is found, $t_0 = 0$, $t_{|\mathcal{I}|} = T$. We define the primal integral $P(T)$ of the run

as:

$$P(T) = \int_{t=0}^{T} p(t)dt = \sum_{i=1}^{|\mathcal{I}|} p(t_{i-1}) \cdot (t_i - t_{i-1}) \tag{3.10}$$

It is evident from the definition of the branch-and-bound algorithm that it poses two competing goals. On one hand, we want to discover better feasible solutions as early as possible. On the other hand, we want to improve on the primal-dual gap to ultimately provide the *certificate of optimality*. The two goals are tackled using separate components, namely, primal heuristics, and branching rules. In the next section, we provide some definitions for the large neighborhood search heuristics relevant to our work. The reader is referred to [16, 46] for an overview of other primal heuristics and [4] for an overview on branching rules. For details on other components in MIP solvers, please refer to [3].

3.2 Primal heuristics

The branch-and-bound algorithm as described in Section 3.1.1 may often take a long time to find good feasible solutions. For this reason, a number of primal heuristics are applied. They range from simpler *rounding heuristics* to more computationally expensive *diving* and *large neighborhood search heuristics*. The large neighborhood search heuristics are improvement heuristics that solve an *auxiliary problem* using branch-and-bound. Here, we give a definition for the general auxiliary problem, which is also a MIP with some modifications to the original MIP. Possible modifications include fixing of variables, additional constraints, and an auxiliary objective function. For an extensive survey on LNS heuristics in MIP, please refer to [47].

Definition 3.14. Auxiliary Problem [47]
Let $\mathcal{P}$ be a given MIP with n variables, and $\mathcal{N}_{orig}$ be the polyhedron defined by the set of its linear constraints. For a polyhedron $\mathcal{N} \subseteq \mathbb{R}^n$ expressed as a set of inequalities and an auxiliary objective function coefficients $\boldsymbol{c}_{aux} \in \mathbb{R}^n$, an auxiliary MIP $\mathcal{P}'$ is defined as:

$$\min\{\boldsymbol{c}_{aux}^{\intercal}\boldsymbol{x} | \boldsymbol{x} \in \mathcal{N}_{orig} \cap \mathcal{N}\} \tag{3.11}$$

The auxiliary problem has the same number of variables as compared to the original problem and the set of feasible solutions of the auxiliary problem is a subset of original feasible solutions. Therefore, each solution for the auxiliary problem is also a valid solution for the original problem. The neighborhood in this case is defined as the polyhedron $\mathcal{N}$. In its most general form, the auxiliary problem can have an auxiliary objective function.

Definition 3.15. Fixing Neighborhood [47]
Let $\mathcal{P}$ be a given MIP with n variables and $\boldsymbol{I}$ be the set of integer variable indices. Given a

subset of integer variables $\boldsymbol{J} \subseteq \boldsymbol{I}$ and an IP-feasible solution $\hat{\boldsymbol{x}} \in \mathbb{R}^n$, the fixing neighborhood is defined as:

$$\mathcal{N}^{fix}(\hat{\boldsymbol{x}}, \boldsymbol{J}) = \{\boldsymbol{x} \in \mathbb{R}^n | x_j = \hat{x}_j \; \forall \; j \in \boldsymbol{J}\} \tag{3.12}$$

It is also common to define an improvement neighborhood that restricts the search space to only improving solutions. Such a neighborhood can be defined by adding an explicit cutoff constraint like in equation (3.13). The cutoff constraint is a function of the primal and dual bound of the MIP.

Definition 3.16. Improvement Neighborhood [47]
Let $\delta \in [0, 1]$. For a given MIP $\mathcal{P}$ with an incumbent solution $\tilde{\boldsymbol{x}}$ and a dual solution $\hat{\boldsymbol{y}}$, an improvement neighborhood is defined as:

$$\mathcal{N}^{imp}(\delta) = \{\boldsymbol{x} \in \mathbb{R}^n | \boldsymbol{c}^\intercal \boldsymbol{x} \leq (1 - \delta) \cdot \boldsymbol{c}^\intercal \tilde{\boldsymbol{x}} + \delta \cdot (\boldsymbol{c}^\intercal \hat{\boldsymbol{y}})\} \tag{3.13}$$

Here, δ controls the minimum improvement that any new incumbent solution must have as compared to the current incumbent solution.

3.2.1 Relaxation Induced Neighborhood Search

Relaxation Induced Neighborhood Search (RINS) is a general purpose primal heuristic for MIP proposed by Danna et al. [26]. The idea is to fix integer variables whose values agree in the LP solution $\hat{\boldsymbol{x}}$ and the incumbent solution $\tilde{\boldsymbol{x}}$.

$$\mathcal{N}_{RINS} = \mathcal{N}^{fix}(\tilde{\boldsymbol{x}}, \boldsymbol{J}) \; \cap \; \mathcal{N}^{imp}(\delta) \tag{3.14}$$

$$where \quad \boldsymbol{J} = \{j \in \boldsymbol{I} | \hat{x}_j = \tilde{x}_j\} \tag{3.15}$$

3.2.2 Crossover

The crossover heuristic implemented in SCIP [42] is inspired from the work of Rothberg [73]. It selects $k \geq 2$ known feasible solutions, $\boldsymbol{X} = \{\hat{\boldsymbol{x}}^1, \ldots, \hat{\boldsymbol{x}}^k\} \subseteq \mathcal{I}$ as reference solutions not necessarily containing the incumbent solution $\tilde{\boldsymbol{x}}$.

$$\mathcal{N}_{Crossover} = \mathcal{N}^{fix}(\hat{\boldsymbol{x}}^1, \boldsymbol{J}) \; \cap \; \mathcal{N}^{imp}(\delta) \tag{3.16}$$

$$where \quad \boldsymbol{J} = \{j \in \boldsymbol{I} | \hat{x}_j^1 = \hat{x}_j^2 = \cdots = \hat{x}_j^k\} \tag{3.17}$$

3.2.3 Mutation

The mutation heuristic implemented in SCIP [42] is also inspired from the work of Rothberg [73]. It fixes a random subset of integer variables to their values in the incumbent solution.

$$\mathcal{N}_{Mutation} = \mathcal{N}^{fix}(\tilde{\boldsymbol{x}}, \boldsymbol{J}) \cap \mathcal{N}^{imp}(\delta) \tag{3.18}$$

where $\boldsymbol{J}$ *is a randomly chosen subset of* $\boldsymbol{I}$

Chapter 4

Supervised Neighborhood Selection for Mixed Integer Programs

4.1 Introduction

Mixed integer program (MIP) solvers commonly utilize large neighborhood search (LNS) heuristics as *improvement heuristics* within the branch-and-bound algorithm. These heuristics improve on a known feasible solution by solving an auxiliary MIP problem, typically a restricted version of the original MIP (see Definition 3.14). As solving a MIP can be computationally expensive, LNS heuristics are executed less often compared to their faster counterparts such as rounding or propagation heuristics. Moreover, the rules of these LNS heuristics are handcrafted and hard-coded into the solver. The static, input-agnostic, and context-independent rules may be unable to adapt to the state of the branch-and-bound search [54]. To this end, we propose a dynamic, data-driven approach to the problem of neighborhood selection.

The idea of exploiting readily available information from the MIP solver to design primal heuristics is not new [36]. Moreover, even traditional LNS heuristics rely on some partial information to prompt their heuristic decisions. For example, in case of Relaxation Induced Neighborhood Search (RINS) [26], the variables that agree in the LP-relaxation solution and the incumbent solution are fixed. Alternatively, we propose to bring together the underlying ideas behind these heuristics such as variable fixing, and intelligently formulate the neighborhood to be searched. Our goal is to automatically infer the heuristic strategy from the data observed on a wide range of problem instances.

For our preliminary studies, we restrict our focus to the variable fixing neighborhood (see Definition 3.15). RINS [26], Crossover [73], and Mutation [73] all use this neighborhood, where the only differing factor is how they decide what variables are to be fixed. Although one can choose the easier way out and just execute all of them, doing so would involve a high computational effort and our mathematical understanding of which heuristic works

best for a given problem is limited. Moreover, any one scheme of fixing variables may be too restrictive for the problem at hand.

The difficulty of the auxiliary problem plays an important role in the performance of the LNS heuristic. Fixing too few variables might make the sub-MIP too difficult to solve, likely reducing the heuristic's success rate. Fixing too many variables might restrict the neighborhood so much that it may not contain any improving solutions. Unfortunately, RINS and Crossover both do not allow for controlling the fixing rate. Mutation is the only neighborhood for which this can be controlled directly. However, fixing a random set of variables is unlikely to be very effective.

We hypothesize that a learning-based framework can help define a neighborhood that would be the most fruitful so that computational resources can be focused on the more difficult parts of the problem. In this chapter, we explore a dynamic and data-driven approach to the design of large neighborhood search heuristics. We propose a supervised neighborhood selection framework for defining a search neighborhood based on variable fixing and then utilize the framework to devise a supervised large neighborhood search (SLNS) heuristic.

4.2 Problem formulation

The common idea behind the LNS heuristics is that they solve an *auxiliary problem* (Definition 3.14) using MIP as a *black-box*. The *large neighborhood* in these heuristics is essentially defined by the search space of the auxiliary problem. An important question we can ask is - how should we define the neighborhood $\mathcal{N}$ at any given node in the search tree? The classical heuristics' approach to this question is through static handcrafted rules. Alternatively, a neighborhood should be defined such that it has a high likelihood for success. Ideally, a neighborhood should be large enough such that it potentially contains many improving solutions but not so large that it can't be searched under reasonable resource limits. Here, we formalize this problem of transforming a given MIP to an auxiliary MIP, as the *neighborhood selection problem*. In the most general form, we can define the neighborhood selection problem as:

Definition 4.1. Neighborhood Selection Problem
Given a MIP $\mathcal{P}$, find an efficient transformation to an auxiliary problem $\mathcal{P}'$. Such that, solving $\mathcal{P}'$ finds the best improving solution for $\mathcal{P}$ within a prescribed node limit v_{max}.

The definition above for the *neighborhood selection problem* is quite broad in scope, encapsulating the many ways a neighborhood can be defined. In this thesis however, we restrict ourselves to the *fixing neighborhood* (Definition 3.15). A reasonable assumption is that fixing more variables will make the auxiliary problem easier and faster to solve. Therefore, the definition of the restricted version of the *neighborhood selection problem* can be given as

Definition 4.2. Neighborhood Selection Problem for Fixing Neighborhood
Given a MIP $\mathcal{P}$ and a known incumbent solution $\tilde{\boldsymbol{x}}$, find the maximal subset of variables that have the same values in the optimal solution $\boldsymbol{x}^*$ as they have in $\tilde{\boldsymbol{x}}$.

4.3 An overview of the framework

We now describe our approach for supervised neighborhood selection for the variable fixing neighborhood. We use a graph attention network (GAT) [78] based supervised learning model to dynamically decide what variables should be fixed in order to define the search neighborhood. The variables to be fixed are predicted based on the variables, constraints and instance characteristics, and the state of the branch-and-bound search. The task is essentially reduced to binary classifcation, where for each variable $j \in \boldsymbol{I}$, we predict the probability that the variable stays fixed at its value in the incumbent solution $\tilde{\boldsymbol{x}}$. In the following sections, we describe in detail our supervised learning task, graph representation for MIP, the GAT model, and how the framework can be utilized in an LNS heuristic.

4.3.1 Supervised learning

For our supervised learning task, we first collect data by running the MIP solver on many diverse problem instances. Hopefully, the solver generates many feasible solutions during the solving process, using the many primal heuristics that are already implemented in the MIP solver. Each time the solver finds a new incumbent solution, we record samples containing the state of the solver $\mathbf{s}^i$, and the new incumbent solution $\tilde{\boldsymbol{x}}^{i+1}$. The set of samples recorded for a given instance a, can be given as $\mathcal{S}_a = \{(\mathbf{s}^i, \tilde{\boldsymbol{x}}^{i+1})\}_{i=0}^{k_a}$ if the solver finds $(k_a + 1)$ improving solutions. We then compute the class labels for each variable $j \in \boldsymbol{I}$ in a *retrospective* manner. Given $(\mathbf{s}^i, \tilde{\boldsymbol{x}}^{i+1}) \in \mathcal{S}_a \; \forall \; i \in \{1, 2, \ldots, k_a\}$, the labels are computed as follows

$$\mathrm{y}_j^i = \begin{cases} 1 & \textit{if} \quad \tilde{x}_j^i == \tilde{x}_j^{i+1} == \cdots == \tilde{x}_j^{k_a+1} \\ 0 & \textit{otherwise} \end{cases} \tag{4.1}$$

Note that the way we define the labels, the resulting neighborhood around an incumbent solution $\tilde{\boldsymbol{x}}^i$, would contain all the improving solutions i.e. $\{\tilde{\boldsymbol{x}}^{i+1}, \ldots, \tilde{\boldsymbol{x}}^{k_a+1}\}$ found by the solver. Moreover, by definition $|\mathbf{y}^i|$ is a monotonically increasing function over $i \in \{1, 2, \ldots, k_a\}$, as a result the neighborhood is naturally relaxed in the beginning and stricter towards the end. We can now define the dataset for an instance a as the set of state-label pairs $\mathcal{D}_a = \{(\mathbf{s}^i, \mathbf{y}^i)\}_{i=1}^{k}$. The complete dataset across all instances in set $\mathcal{A}$ can be given as

$$\mathcal{D} = \bigcup_{a \in \mathcal{A}} \mathcal{D}_a \tag{4.2}$$

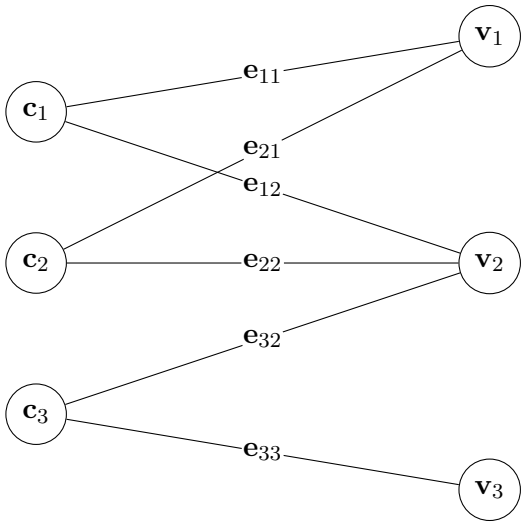


Figure 4.1: Graph representation of a mixed integer program

We now define our supervised learning task as binary classification where given a state-labels pair $(\mathbf{s}, \mathbf{y}) \in \mathcal{D}$, we learn a function $\mathcal{F}_\theta(\mathbf{s})$ parametrized by θ that outputs the probabilities for the labels to be 1 or 0. We train the classifier model by minimizing the weighted cross entropy loss function given below

$$\mathcal{L}(\theta) = \sum_{(\mathbf{s},\mathbf{y})\in\mathcal{D}} \left\{ -\frac{1}{|\boldsymbol{I}|} \sum_{j\in\boldsymbol{I}} \rho \mathrm{y}_j \log \mathrm{z}_j + (1-\mathrm{y}_j) \log\left(1-\mathrm{z}_j\right) \right\} \tag{4.3}$$

Here, y_j are the true labels, and z_j are the probabilities for each variable j in the sample $(\mathbf{s}, \mathbf{y})$. The set $\boldsymbol{I}$ denotes the set of integer variables for the instance the sample belongs to. The variable ρ controls the weight for class 1, and is defined as

$$\rho = \frac{\sum_{j\in\boldsymbol{I}}(1-\mathrm{y}_j)}{\sum_{j\in\boldsymbol{I}} \mathrm{y}_j} \tag{4.4}$$

4.3.2 Graph representation

A natural way to represent a MIP problem $\mathcal{P}$ is using the constraint-variable bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{C}, \mathcal{E})$, see Figure 4.1. Here $\mathcal{V}$ is the set of variable nodes, $\mathcal{C}$ is the set of constraint nodes, and $\mathcal{E}$ is the set of edges, where an edge between a variable and constraint exists only if it has a nonzero coefficient in the constraint. Gasse et al. [38] proposed using this representation to encode the state of the MIP solver's branch-and-bound process. This representation is ideal as it is scale-independent, permutation-invariant, and computationally efficient for sparse problems typically encountered in practice. The graph representation also naturally captures the dependencies between the variables and constraints.

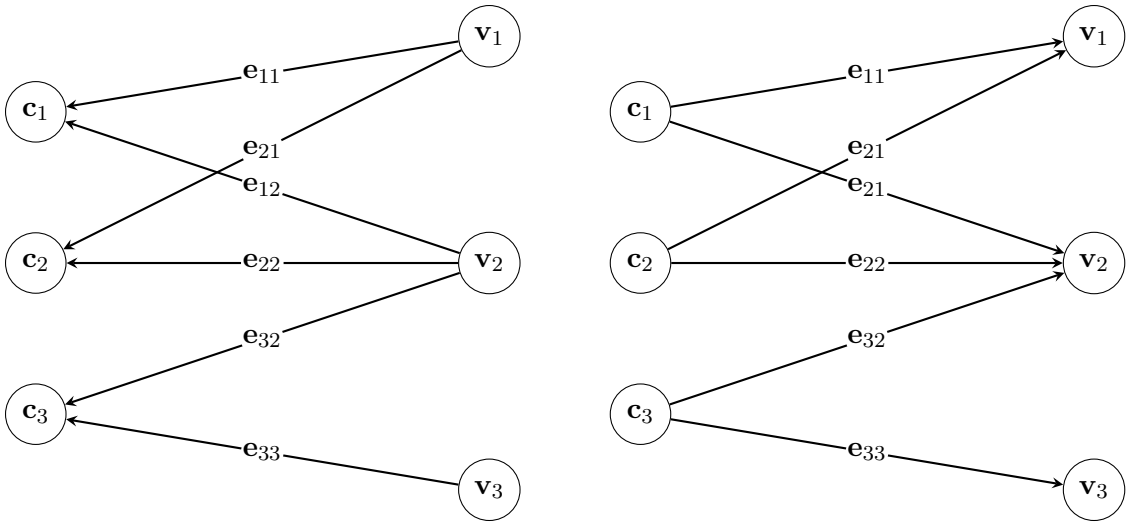


Figure 4.2: Our graph attention model

4.3.3 Graph attention network

Now, we describe our graph attention network model [78] that attempts to approximate the function $\mathcal{F}_\theta(\mathbf{s})$. Our model is an adaptation of the graph convolutional network based model proposed by Gasse et al. [38] with added *self-attention* layers. Self-attention layers, essentially, learn to prioritize the features of nearby nodes, by implicitly learning different weights for each node in the neighborhood. The main reason behind using a graph attention model is that it is specifically designed for *inductive tasks*, where the graph structures can vary. The state $\mathbf{s}$ of the MIP solver's branch-and-bound process can be encoded in the graph as $\mathbf{s} = (\mathcal{G}, \mathbf{V}, \mathbf{E}, \mathbf{C})$. Here $\mathcal{G}$ is the bipartite graph as defined above, $\mathbf{V} \in \mathbb{R}^{n \times \mathrm{d}}$ is the feature matrix for the variables, $\mathbf{E} \in \mathbb{R}^{m \times n \times \mathrm{e}}$ is a (sparse) tensor of edge features, and $\mathbf{C} \in \mathbb{R}^{m \times \mathrm{c}}$ is the feature matrix for the constraints. For the details about features, please refer to Appendix A. Our GAT model performs two message passing operations, one from variables to constraints, and another from constraints to variables, see Figure 4.2. The operations performed by the GAT (See Figure 4.3) are as follows

$$\mathbf{c}_i \leftarrow f_{\mathcal{C}}(\mathbf{c}_i, \sum_{j}^{(i,j)\in\mathcal{E}} \alpha_{ij} \cdot g_{\mathcal{C}}(\mathbf{v}_j)), \qquad \mathbf{v}_j \leftarrow f_{\mathcal{V}}(\mathbf{v}_j, \sum_{i}^{(i,j)\in\mathcal{E}} \beta_{ij} \cdot g_{\mathcal{V}}(\mathbf{c}_i)) \tag{4.5}$$

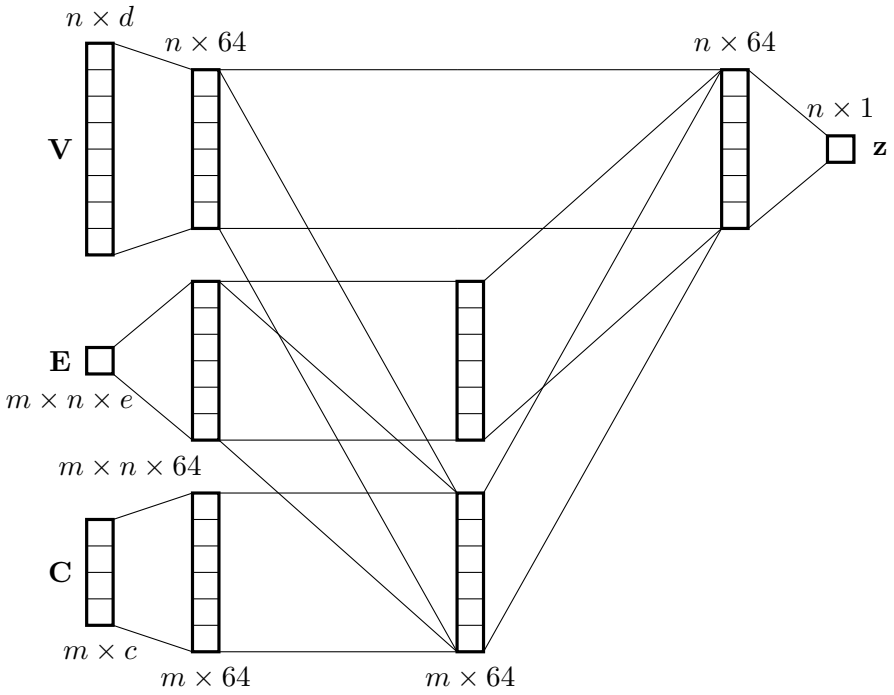


Figure 4.3: Graph convolution operations

For each $i \in \mathcal{C}$, and $j \in \mathcal{V}$, where $f_{\mathcal{C}}$, $f_{\mathcal{V}}$, $g_{\mathcal{C}}$, and $g_{\mathcal{V}}$ are 2 layer perceptrons with $ReLU$ activation functions. The attention coefficients α_{ij}, and β_{ij} are defined as

$$\alpha_{ij} = \frac{\exp(h_{\mathcal{C}}(\mathbf{c}_i, \mathbf{e}_{ij}, \mathbf{v}_j))}{\sum_{(i,j)\in\mathcal{E}} \exp(h_{\mathcal{C}}(\mathbf{c}_i, \mathbf{e}_{ij}, \mathbf{v}_j))} \qquad \textit{and} \qquad \beta_{ij} = \frac{\exp(h_{\mathcal{V}}(\mathbf{c}_i, \mathbf{e}_{ij}, \mathbf{v}_j))}{\sum_{(i,j)\in\mathcal{E}} \exp(h_{\mathcal{V}}(\mathbf{c}_i, \mathbf{e}_{ij}, \mathbf{v}_j))} \tag{4.6}$$

Here, $h_{\mathcal{C}}$ and $h_{\mathcal{V}}$ are dense networks with one node and LeakyReLU activation function with a negative slope coefficient of 0.2. The graph attention network uses graph convolution operations to aggregate the features of a node's neighbors. The features for constraint, variables, and edges are all mapped to a vector of fixed size, in our case 64, to make these operations easier. After the convolution operation, the variable features would essentially include information from every neighboring constraint node and every variable node at a distance 2 from itself. Finally, the updated variable features are passed through another 2-layer perceptron and the sigmoid function to obtain the probability estimate for the variable to be fixed.

4.4 Supervised large neighborhood search

Here, we describe how we use the neighborhood selection framework for our supervised large neighborhood search heuristic. While solving a MIP problem, we can extract the variable-

constraint bipartite graph along with the features from the MIP solver to get the state encoding, $\mathbf{s} = (\mathcal{G}, \mathbf{V}, \mathbf{E}, \mathbf{C})$. The graph attention model takes the state encoding as input and estimates the probabilities for variables to be fixed.

$$\mathbf{z} = \mathcal{F}_\theta(\mathbf{s}) \tag{4.7}$$

The estimated probabilities can then be used to devise a fixing strategy for defining the search neighborhood. A possible strategy is to define a decision boundary by setting a probability threshold $\phi_{threshold} \in [0, 1]$, thereby fixing any variable j in $\boldsymbol{I}$ with a probability greater than $\phi_{threshold}$. The set of fixing variables can be defined as

$$\boldsymbol{J} = \{\ j \mid \mathrm{z}_j \geq \phi_{threshold}\} \tag{4.8}$$

Another strategy, is to set a fixing rate $\phi_{fixing} \in [0, 1]$ instead. In this case, we sort the variables according to their estimated probabilities from high to low, and select the top $\phi_{fixing} \cdot |\boldsymbol{I}|$ variables.

4.5 Data collection

To collect training data for our graph attention model, we record the state of the MIP solver every time a new incumbent solution is found. The state of the solver at any time can be characterized by the LP-relaxation solution, the incumbent solution, and the branch-and-bound history. Here, we discuss details about the features and labels that we record. A complete list of features used can be found in Appendix A.

4.5.1 Features

The ideal set of features for our problem must capture critical information about the structure of the MIP instance, as well as the state of the branch-and-bound search. The features must also be computationally efficient to extract. Previous work [53, 54, 38, 30] have identified many different features for constraints and variables. Ding et al. [30] classify these features as *basic features*, *LP features*, and *structure features*. Although it should be possible for a graph network to learn complex relations from just the basic features, they are known to have difficulty in learning simple counting operations. For this reason, Ding et al. found it best to include both LP features and structure features. We augment the set of features with *History or Branch-and-bound* features that include historical information like incumbent solution and average incumbent solution. Overall, our features can be extracted in linear time with respect to the number of nonzeros in the problem instance.

Feature scaling is a common technique used to stabilize the learning process of a neural network, as it mitigates the problem of vanishing or exploding gradients. However, scaling features without any consideration to the problem at hand can cause loss of information.

To address this issue, we appropriately scale each individual feature trying to preserve the problem structure as well as we can. For example, the objective function coefficients are scaled by dividing by its L2 norm. Another example is the number of constraints for each variable, where we scale the feature by dividing it by the total number of constraints. The reason this works well is because presolving takes care of the many redundancies in the problem instance. In situations where a natural way of scaling is not apparent, we use the query based standardization similar to [53].

4.5.2 Labels

The labels for each integer variable, are computed using the retrospective labeling scheme as described in Section 4.3.1. After we have solved an instance, we know the set of improving solution found during the search. We then go back and compute the labels for each variable across the collected samples. The label is 1 if the variable assumes the same value in the incumbent solution and all the improving solutions. Otherwise, it is labelled as 0.

Note that Ding et al. [30] propose a similar strategy, however, their approach is designed to directly predict the solution values for binary variables. They also define the terminology of *stable* and *unstable* variables. Using the same terminology, we are trying to predict which variables are stable as opposed to predicting their solution value.

4.6 Empirical analysis

We performed extensive experimentation to evaluate our graph attention network model for neighborhood selection and the ensuing SLNS heuristic. In the following sections, we describe in detail the setup, the dataset, key aspects of data collection, and experiments, as well as summarize our computational results.

4.6.1 Setup

Our framework comprises of many interacting components. For data collection and evaluation, we modify SCIP 6.0.1 [42] and use IBM ILOG CPLEX 12.9.0 [49] as the underlying LP solver. Overall, the framework is written in Python 3.6.8 using PySCIPOpt [67], a Python interface for SCIP, and TensorFlow 1.12.0 [1] for implementing the graph attention model. The framework was adapted from the work of Gasse et al. [38] on branching strategies. All data collection tasks and experiments were run on the Compute Canada Cedar cluster composed of nodes with the following CPUs:

1. Intel E5-2683 v4 Broadwell @ 2.1Ghz
2. Intel E7-4809 v4 Broadwell @ 2.1Ghz
3. Intel E5-2650 v4 Broadwell @ 2.2GHz

4. Intel Platinum 8160F Skylake @ 2.1Ghz

No restriction was placed on the nodes' CPU architecture for data collection tasks, however to avoid performance variability in experiments, we restrict the architecture to *Skylake*. The graph attention model was trained on the same cluster using NVIDIA P100 Pascal GPU with 12GB Memory. For performance evaluation tasks, we run each solver process on a dedicated CPU core, minimizing the effects of shared resources among multiple processes. We allow presolving and cutting plane separation at the root node, and disable presolving restarts, as is common practice in previous studies [53, 38]. All SCIP parameters are kept at their default values, unless noted otherwise.

4.6.2 Instances

To test the effectiveness and generality of our neighborhood selection framework, we use MIP instances from the "Collection" and "Benchmark" sets from MIPLIB2017 [41]. The "Benchmark" set contains 240 instances specifically selected for benchmarking purposes whereas the "Collection" set is larger and much more diverse. We reserve the "Benchmark" instances to serve as our test set. For the training dataset, we select comparable instances from the "Collection" set (ones that are not in "Benchmark" set) that are tagged as "benchmark_suitable". To avoid memory issues, we put a hard cap on the number of nonzeros ($\leq 500,000$). We also remove any *feasibility instances* (without an objective function), *infeasible instances*, and *unbounded instances*. After filtering, we have 204 instances in the training set and 181 instances in the test set. We further use an 80/20 split for training and validation, respectively. Detailed information about the instances can be found on the MIPLIB website, `https://miplib.zib.de`.

4.6.3 Data collection

For the data collection, we implement an *event handler* plugin for SCIP [42]. The plugin is called every time SCIP finds a new incumbent solution, at which point we capture the state of the solver and the new incumbent solution. To be precise, we only start capturing data samples after the presolving and cutting plane separation is complete, and the root LP-relaxation is solved. The captured raw data samples are later used to compute the variable and constraint features, and to perform the retrospective labelling of the variables. Note that a new incumbent may be found by one of the many primal heuristics implemented in the solver or a leaf node in the search tree. As the LP solution and incumbent solution are the same at the leaf node, the resulting data samples do not add much in terms of capturing the state of the solver. Therefore, we use these samples only for the labelling of variables but exclude them from being used for training.

In order to collect the training and test data, we run the solver on the respective instances for one hour each. It is unreasonable to expect every instance to be solved within this time

limit or even to find many incumbent solutions. However, we are able to collect a large enough dataset even though some instances may not be represented in the training or test data. Note that it is also possible that some instances find many more solutions than others and therefore result in more samples.

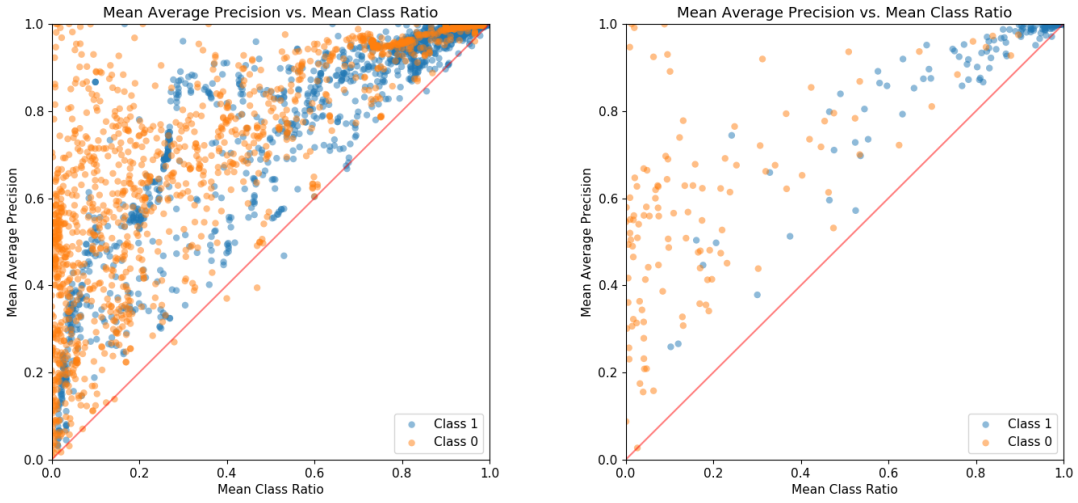
4.6.4 Training

We train our graph attention model on the collected training samples using the Adam optimizer with a learning rate of 0.001. We use a mini-batch size of one which means that we compute the gradients on the graph from one sample at a time. As we can have a different number of samples from each instance, we resample the data such that each instance has equal representation in the training data.

We use a patience of 10 epochs i.e. we let the training continue for 10 epochs even if there is no improvement after which, we decrease the learning rate by a factor of 0.2. We stop the training early if there is no improvement on the validation loss for 20 epochs. The training converges within 10-20 epochs with minor improvements later on and takes about 2 hours to complete on the NVIDIA P100 Pascal GPU.

4.6.5 Computational results

Now, we describe our computational results across different experiments designed to evaluate our graph attention model and its utility within an LNS heuristic.



(a) For each sample in the dataset

(b) Averaged for each instance in the dataset

Figure 4.4: Average precision vs. Class ratio

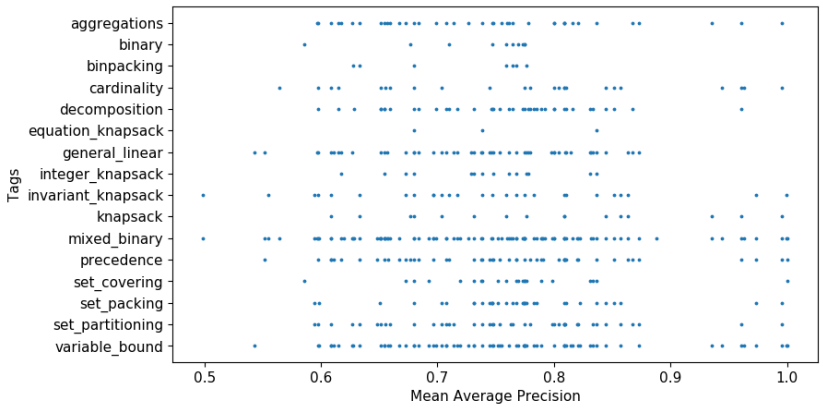


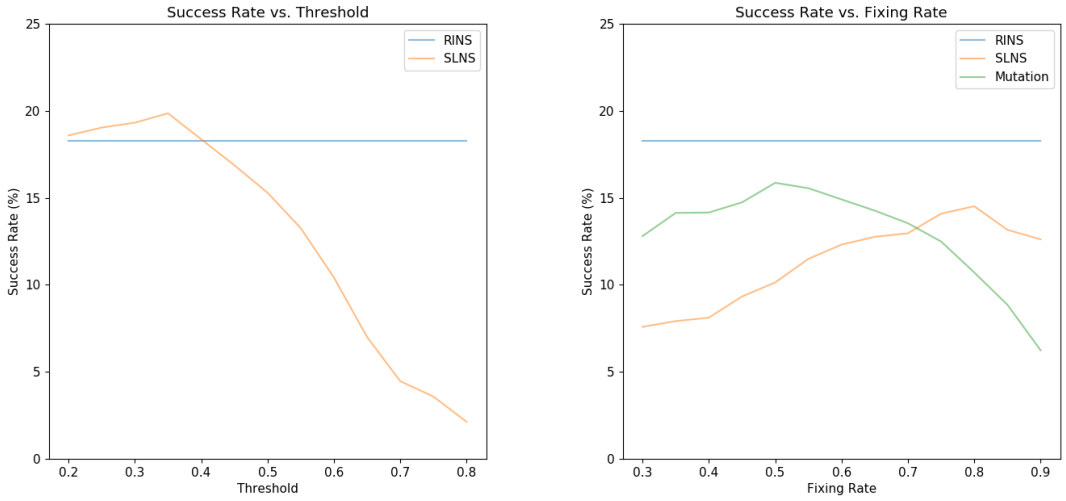
Figure 4.5: Mean average precision vs. Tags

Predicting class labels

Prediction of class labels as per the classification task described in Section 4.3.1 is difficult due to the high imbalance and high variability in the two classes across collected samples, even when the samples come from the same instance. As accuracy is not the perfect measure to capture the quality of predictions in this case, we use the average precision metric [82] similar to Ding et al. [30]. Average precision, essentially, is the area under the curve of the precision-recall curve and measures how well a classifier ranks items from one class above items from the other. Note that the average precision metric is asymmetric for the two classes, therefore, we report the metric for both the classes. Figure 4.4a, shows the average precision metric vs. the class ratio for each sample in the dataset. Figure 4.4b, on the other hand, shows the mean average precision vs. mean class ratio per instance. We can say that the classifier achieves good performance as the average precision in most cases is considerably greater than the class ratio. Note that a random classifier would achieve an average precision close to the class ratio. Table 4.1, shows the mean average precision and mean class ratio across all instances in the dataset. To see if our model performs well on certain type of instances, we also plot the mean average precision vs. the tags (Figure 4.5) assigned to the instances in the MIPLIB2017 [41]. The tags, essentially, characterize the structure of the instances. Unfortunately, we did not find any correlation.

	Mean Class Ratio	Mean Average Precision
Class 1	0.81	**0.91**
Class 0	0.19	**0.58**

Table 4.1: Mean average precision for class 1 and class 0 variables



(a) Success rate vs. Threshold

(b) Success rate vs. Fixing rate

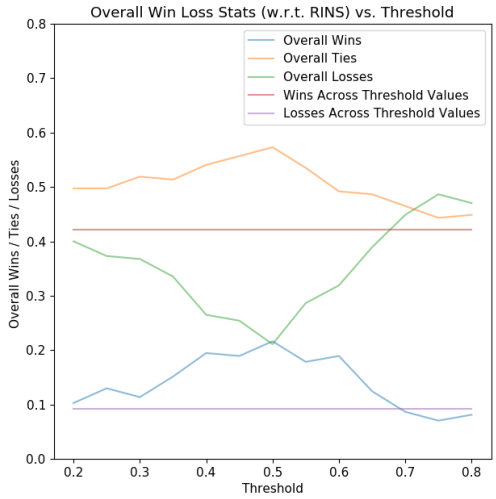
Figure 4.6: Success rate of SLNS

Fixing strategy

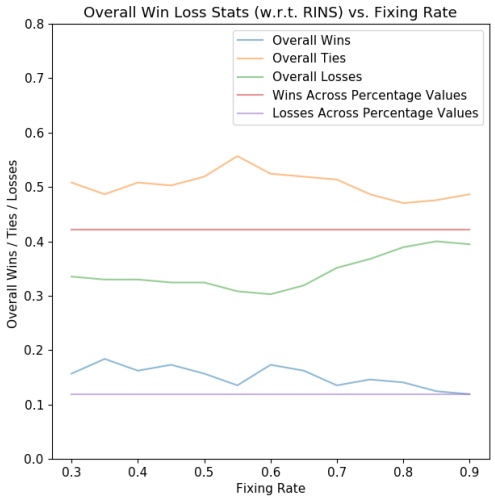
In the previous section, we described how our graph attention model performs well in terms of separating the two classes. Now, we design an experiment to determine which of the two strategies described in Section 4.4 works best. We test the two strategies, with choice of threshold in the range (0.2, 0.8), and choice of fixing rate in the range (0.3, 0.9) with steps of 0.5. Figure 4.6, shows the success rate of SLNS with respect to different values of threshold and fixing rate. For reference, we plot the success rate for RINS, that has a fixing rate determined by the LP-relaxation solution and the incumbent solution. Finally, in the case of fixing rate, we also compare to the Mutation neighborhood. Figure 4.6a shows that the success rate peaks at a value of 0.35, at which point our heuristic has a higher success rate compared to RINS. As expected, the success rate decreases with increasing threshold as we fix less and less variables. Figure 4.6b shows the success rate vs. fixing rate. Again as expected, we see a rise in success rate as the fixing rate increases and then a drop thereafter.

SLNS vs. RINS

Next, we study how SLNS fares against RINS, as our heuristic most directly compares with it. RINS is one of the better performing LNS heuristics and both SLNS and RINS can run in exactly identical situations. Other heuristics, like Crossover cannot execute until at least two or more solutions are available. For this experiment, we run both SLNS and RINS at each node in stealth mode [54], where any new incumbents found do not replace the existing incumbent solution. We report the number of *overall wins*, *ties*, and *overall losses* for SLNS as compared to RINS. An overall win is where SLNS finds an incumbent solution that is



(a) Overall win/loss/tie vs. Threshold



(b) Overall win/loss/tie vs. Fixing Rate

Figure 4.7: Overall win/loss/tie for SLNS with respect to RINS

better than the solution found by RINS. An overall loss is defined analogously, and a tie is when both heuristics find a solution with the same primal bound or both are unable to find a solution. In figures 4.7a and 4.7b, we can see that the number of absolute losses for SLNS compared to RINS are always higher than the number of absolute wins, except in the case of 0.5 threshold where it comes quite close. We believe that this is due to the high variability in classes across the different problem instances, and any one value of threshold or fixing rate cannot capture all scenarios. The red line in the figure, denotes at least one win across the different threshold or fixing rate values. The purple line in the figure, denotes that it was a loss for all threshold or fixing rates. This means that if we use an adaptive fixing rate we can achieve significantly more wins. Therefore, to address this issue, we use an adaptive fixing rate similar to [47].

SLNS Performance

In our final experiment, we evaluate the performance of SLNS as compared to the default set of heuristics that are a part of SCIP [42]. All SCIP settings were kept at their default values. We run the solver on every instance in the test set, once with SLNS enabled, and once without SLNS. We set a timelimit of one hour for each run. We execute the experiment with five different seeds to control for the performance variability [64]. For a total of 181 test instances this results in 905 runs. From this, we filter any runs where no incumbent solution was found beyond the root node, or where the gap was infinite. After filtering, we get a total of 728 runs for which we report the data in Table 4.2. The table shows the comparison between the two conditions. In particular, we report the shifted geometric means (with a

shift of 1) for the primal gap, primal dual gap, primal integral, time to best incumbent, total time, and total nodes, and the raw values of number of instances solved and number of times an overall better solution was found as compared to the other condition. We can see that there is a significant improvement, especially in primal bound and primal integral when our heuristic is added to the existing pool. Moreover, in almost half the cases a better solution was found when SLNS was enabled. Another key finding is that when both SLNS and RINS are enabled, SLNS is on average five times more likely to find an improving solution as compared to RINS. We report exhaustive results for this experiment in Table B.1 and Table B.2 in Appendix B.

	without SLNS	with SLNS	Improvement (%)
Primal Gap (%)	4.60	**3.68**	**20.00**
Primal Dual Gap (%)	13.01	**12.23**	**6.00**
Primal Integral	21.86	**17.05**	**28.21**
Time to Best Incumbent (s)	**546.39**	549.00	-0.48
Total Time (s)	1464.23	**1394.02**	**4.80**
Total Nodes	**11292.12**	14899.34	-31.94
Number of Instances Solved	240.00	**256.00**	**6.66**
Number of Times Better Solution	75	**341**	**354.66**

Table 4.2: Performance comparison with and without SLNS (Total 728 runs)

4.7 Summary

In this chapter, we formalized the idea of neighborhood selection problem in the context of large neighborhood search heuristics. We presented our graph attention network based supervised learning framework for predicting the ideal variable fixing neighborhood. Finally, we devised a supervised large neighborhood search heuristic based on our framework. We evaluated our supervised neighborhood selection framework and our heuristic through multiple experiments. The experiments show that our method is promising as compared to the traditional LNS heuristics. Our heuristic improves the solver's performance on many key metrics, primarily the primal bound and the primal integral, which are frequently used to judge the quality of a heuristic. Moreover, our heuristic is also five times more likely to succeed as compare to RINS. Future work in this direction would help uncover more insights.

Chapter 5

MIP-based Primal Heuristic for the Linear Ordering Problem

5.1 Introduction

The linear ordering problem (LOP) is defined as follows. Let $G = (V, A)$ denote a complete directed graph or digraph on n vertices, where $V = \{1, 2, \ldots, n\}$ is the set of vertices, A is the set of arcs that includes (i, j) and (j, i), for every distinct pair of vertices i and j in V, and each arc $(i, j) \in A$ has a weight c_{ij}. A *linear ordering* of the vertices $\{1, 2, \ldots, n\}$ is denoted by $\langle v_1, v_2, \ldots, v_n \rangle$, where v_1 precedes v_2, v_2 precedes v_3, and so on. The objective of the linear ordering problem is to find an ordering σ such that $\sum_{i,j:\sigma(i) \prec \sigma(j)} c_{ij}$ is maximized, where $\sigma(i) \prec \sigma(j)$ denotes that vertex i precedes vertex j in the linear ordering σ.

LOP is an NP-hard problem [37], closely related to other problems in graph theory, such as the *feedback arc set problem*, the analogous *feedback node set problem*, as well as the *node induced acyclic subdigraph problem*. The problem is relevant to voting theory [50], where an aggregated ranking is to be computed from individual preferences such that it most appropriately represents the preferences. This is also relevant to the ranking of a set of teams in sports tournaments. The *input-output analysis* in the field of economics and the problem of machine scheduling under precedence constraints are other problems that can be modeled using LOP. For a detailed discussion on the applications of the linear ordering problem, the reader is referred to [68].

As LOP can model many problems of practical importance, it has garnered a lot of attention in the literature. An exact method for LOP includes formulating it as a 0-1 integer program and then solving it using the branch-and-bound algorithm [70]. However, as the computation time can grow rapidly with the size of the problem, it becomes impractical for problems of large sizes. Population based metaheuristic methods such as the genetic algorithm, and scatter search, have been successfully applied to LOP [68]. However, the current state-of-the-art method for LOP is the memetic algorithm that combines the genetic algorithm with local search [74]. Ceberio et al. [22] propose a more efficient way for local

search in the context of LOP that excludes provably suboptimal solutions from the search neighborhood. Hybrid approaches that combine heuristic and exact methods have also been developed for the LOP [51]. They propose a MIP heuristic that generates a starting feasible solution based on the solution to the LP relaxation. Using the starting solution, they then define a neighborhood which is then solved optimally using a MIP solver.

As compared to the existing MIP heuristic, our heuristic relies on partitioning the set of vertices S into an ordered pair of subsets (S_1, S_2) such that the difference between the weights of all arcs from S_1 to S_2 and the weights of all arcs from S_2 to S_1 is maximized. The set of vertices are recursively partitioned until we can quickly solve LOP instances on each subset using a MIP solver. The solution to the original LOP instance is then constructed by concatenating the solutions to the LOP instances on each subset. In comparison with MIP heuristic, our heuristic is fast and generates good solutions close to the optimal and hence can be used as a primal heuristic in branch-and-bound.

5.2 Problem formulation

A linear ordering problem on graph $G = (V, A)$ with arc weights $c_{ij}\ \forall\ (i,j) \in A$, can be formulated as a 0-1 integer programming problem. We define a binary decision variable x_{ij} for each arc $(i,j) \in A$ such that:

$$x_{ij} = \begin{cases} 1, & \text{if } i \prec j, \text{ vertex } i \text{ precedes vertex } j \\ 0, & \text{otherwise} \end{cases}$$

The canonical Integer Linear Programming formulation for the LOP [68] can be given as follows:

$$\text{Maximize} \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{5.1}$$

s.t.

$$x_{ij} + x_{ji} = 1 \qquad \forall\ i,j \in V : i < j \tag{5.2}$$

$$x_{ij} + x_{jk} + x_{ki} \leqslant 2 \qquad \forall\ i,j,k \in V : i < j, i < k, j \neq k \tag{5.3}$$

$$x_{ij} \in \{0,1\} \qquad \forall\ i,j \in V : i \neq j \tag{5.4}$$

The objective function (5.1) maximizes the total weight of all arcs (i,j) such that $i \prec j$. Constraint (5.2) ensures that either $i \prec j$ or $j \prec i$, but not both. Constraint (5.3) prohibits a directed cycle where $i \prec j$, $j \prec k$, and $k \prec i$. Constraint (5.4) constrains the variables x_{ij} to take values in the set $\{0,1\}$.

5.3 Partitioning Problem

We design the primal heuristic for the linear ordering problem based on the concept of *strongly connected components* in graph theory. A directed graph is said to be *strongly connected* if every vertex in the graph is reachable from every other vertex. The *strongly connected components* of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected. If every strongly connected component in the graph is contracted to a single vertex, the resulting graph is a *directed acyclic graph*. The linear ordering problem itself is based on a complete directed graph which is strongly connected. However, if we had some way of determining edges at least some of which participate in the optimal solution, we can compute the strongly connected components over this new graph. The resulting directed acyclic graph can then be used to establish the precedence relation between the vertices that belong to the different strongly connected components. The linear ordering problem, as a result, can be decomposed into subproblems on each of the strongly connected components.

To see how this could be useful, consider the graph $G = (V, A)$ for the linear ordering problem. We form a new graph G' by greedily selecting the edges (i, j) between each pair of vertices i and j, such that $c_{ij} > c_{ji}$. The strongly connected components of graph G' form an acyclic graph $S = (V_s, A_s)$ where $V_s = \{s_1, s_2, \ldots, s_k\}$ is the set of components and A_s is the set of arcs between the components. We can then define the precedence relation between the different components as $s_i \prec s_j$ if $(s_i, s_j) \in A_s$. Consequently, we can also define the precedence relation between the vertices belonging to the different components as $u \prec v$ if $u \in s_i, v \in s_j$ and $s_i \prec s_j$. Once these precedence relations are established, we just need to find the precedence relations between the vertices within each component to solve the complete linear ordering problem.

In practice, however, it is too much to expect that the graph resulting from the greedy selection of edges would contain more than one component. So we devise an IP to find the best partition to divide the linear ordering problem into subproblems.

5.3.1 Formulation as an integer programming problem

For a given linear ordering problem, the *partitioning problem* splits the set of vertices V into two subsets S_1 and S_2 such that $\sum_{i \in S_1, j \in S_2}(c_{ij} - c_{ji})$ is maximized. It is formulated as a 0-1 integer programming problem. We define a decision variable y_i for each vertex $i \in V$ such that:

$$y_i = \begin{cases} 1, & \text{if vertex } i \text{ belongs to } S_1 \\ 0, & \text{if vertex } i \text{ belongs to } S_2 \end{cases}$$

The Integer Linear Programming formulation for the partitioning problem can then be given as:

$$\text{Maximize} \sum_{(i,j)\in A} c_{ij}(y_i - y_j) \tag{5.5}$$

s.t.

$$y_i \in \{0,1\} \qquad \forall\, i \in V \tag{5.6}$$

Algorithm 1: Partitioning Problem for the Linear Ordering Problem

Input : Weight matrix $[C]_{n\times n}$, and LP relaxation solution $[P]_{n\times n}$
Output: Subsets (S_1, S_2) of the set of vertices V in graph $G = (V, A)$

```
for each x_ij == 1 in P do
    add constraint (y_i − y_j ≥ 0) to LOP-Partition formulation;
end
Solve the LOP-Partition (with the added constraints);
Let S_1 be the set of all vertices with y_i == 1;
Let S_2 be the set of all vertices with y_i == 0;
Return (S_1, S_2);
```

We call this problem LOP-Partition. The objective function (5.5) maximizes the difference between the total weight of the arcs from subset S_1 to S_2 and the total weight of the arcs from S_2 to S_1. The only set of constraints for the problem, Constraint (5.6), states that the decision variables take values from the set $\{0, 1\}$.

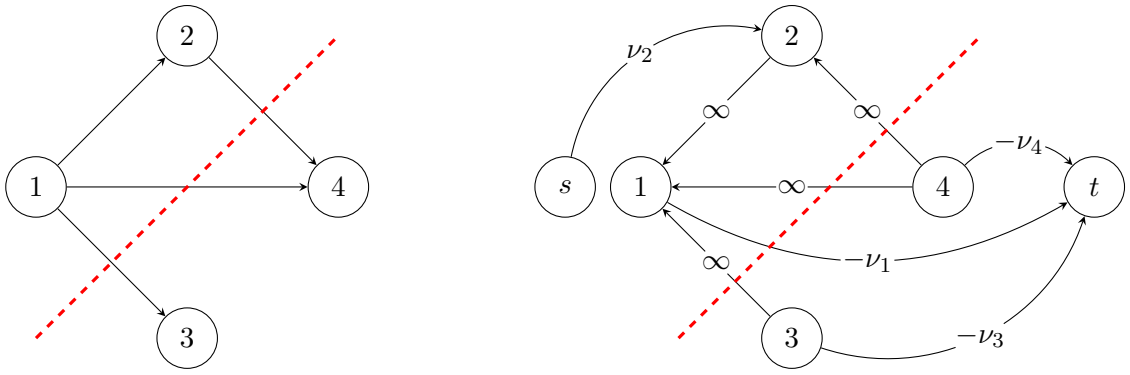
We can see that the problem is trivial when we have no constraints. However, when using branch-and-bound to solve the LOP, we have the solution to the LP relaxation at each node where some variables x_{ij} have been fixed to either 0 or 1. For all variables x_{ij} set to 1, we introduce additional precedence constraints in the above problem to ensure that we adhere to the partial solution. The constraint can be given as follows:

$$y_i - y_j \geq 0 \tag{5.7}$$

5.3.2 Formulation as a minimum cut problem

For a given LOP-Partition, we compute the graph H exhibited by the precedence constraints in (5.7). By definition of LOP, this graph is expected to be a directed acyclic graph (DAG). We then compute the potentials for each node j in H as $\nu_j = \sum_{j\in V, j\neq i}(c_{ji} - c_{ij})$. The goal then is to maximize $\sum_{j\in V} \nu_j$ subject to the precedence constraints of the DAG H. This problem as is it turns out, is equivalent to the *restricted primal* of the Hitchcock problem [71]. We solve it using the min-cut problem.

1. We compute the transitive closure of H and reverse the edges to compute H'. The capacity of each edge in H' is set to ∞.



Graph H Graph H'

Figure 5.1: Formulation as a minimum cut problem

2. We then add a super source s and connect it to all nodes j where $\nu_j > 0$ and set the edge capacities to ν_j.

3. Similarly, we add a super sink t and connect all nodes j where $\nu_j < 0$ to t and set the edge capacities to $-\nu_j$.

We then compute the min-cut for the graph H'. All the nodes reachable from s belong to the set S_1 and all the nodes reachable from t belong to the set S_2. The reason this method works is that maximizing $\sum_{j \in V} \nu_j$ subject to the precedence constraints is equivalent to maximizing $\sum_{j \in S_1} \nu_j + \sum_{j \in S_2} -\nu_j$ or minimizing $\sum_{j \in S_2} \nu_j + \sum_{j \in S_1} -\nu_j$ which is exactly what the min-cut computes.

5.4 Algorithm

The heuristic recursively partitions the set of vertices in the graph $G = (V, A)$ into two subsets. The partitioning is performed until the subsets can be efficiently solved using a MIP solver. If the subsets are sufficiently small, we use the MIP solver to solve them optimally. The pseudocode for the heuristic is given in Algorithm 2.

5.4.1 Improvement Phase

The linear ordering $\langle v_1, v_2, \ldots, v_n \rangle$ obtained via the primal heuristic as described in Algorithm 2 serves as a starting solution for improvement heuristics. We use local search to look for *insert* moves that further improve the objective function (5.1) [58]. We perform the move that causes maximum improvement until no improvement is possible using such a move.

5.4.2 Node Selection

When using the branch-and-bound search in IP, we may have multiple nodes in the queue waiting to be processed. For each node, we have the solution to the LP relaxation and the

Algorithm 2: Primal Heuristic to obtain Integer Feasible Solution

Input : Weight matrix $[C]_{n\times n}$, and LP relaxation solution $[P]_{n\times n}$
Output: An ordering $\langle v_1, v_2, \ldots, v_n\rangle$ of the vertices in graph $G = (V, A)$

if $n \leq 40$ **then**
  return optimal ordering v^* using a MIP solver;
end
Run Algorithm 1 (partitioning problem) to obtain subsets S_1 and S_2;
Find the weight matrix $[C_{S_1}]_{|S_1|\times|S_1|}$ for subset S_1;
Find the weight matrix $[C_{S_2}]_{|S_2|\times|S_2|}$ for subset S_2;
Find the LP solution matrix $[P_{S_1}]_{|S_1|\times|S_1|}$ for subset S_1;
Find the LP solution matrix $[P_{S_2}]_{|S_2|\times|S_2|}$ for subset S_2;
Run Algorithm 2 with $[C_{S_1}]$ and $[P_{S_1}]$ to get ordering v^1 of S_1;
Run Algorithm 2 with $[C_{S_2}]$ and $[P_{S_2}]$ to get ordering v^2 of S_2;
Obtain v by concatenating v^1 and v^2;
Return v;

branching variable at its parent node, and the branching direction. This information is used to provide the partial solution to the heuristic and a node with the best heuristic solution is chosen to be processed next.

5.5 Empirical analysis

The computational experiments were conducted on an Intel Core i7-7700HQ with 2.80 GHz 64-bit processor, 8.0 GB of RAM and Ubuntu 16.04 64-bit as the Operating System. The heuristic was implemented in C++. The LP and MIP problems were solved using CPLEX 12.8 on a single thread. The experiments on the difficult instances were run under the time restriction of 600 seconds however no such constraint was placed on the easier instances.

5.5.1 Dataset

The data set for the computational experiments was obtained from the Optsicom project [69]. We use the following data for our experiments.

Special Instances:

These are problem instances that were used in publications. The *EX* instances were used in [23]. The *ATP* instances were created from the results of ATP tennis tournaments in 1993/1994. Nodes correspond to a selection of players and the weight of an arc (i, j) is the number of victories of player i against player j.

Table 5.1: Computational Results for Special Instances

		CPLEX		H		H+NS	
Instance	Size	NLPS	CPU(s)	NLPS	CPU(s)	NLPS	CPU(s)
ATP66	66	113	56	57	41	62	44
ATP76	76	1046	1147	651	778	254	368
EX4	50	133	22	97	21	128	46
EX5	50	197	35	183	45	123	42
EX6	50	47	10	37	12	58	18

Table 5.2: Computational Results for RandomAI Instances

		Best Known [69]			H		H+NS	
Instance	Size	Solution	Bound	Gap(%)	Solution	Gap(%)	Solution	Gap(%)
t1d100.01	100	106852	114468	7.128	106288	7.696	106344	7.639
t1d100.02	100	105947	114077	7.674	104905	8.743	104952	8.694
t1d100.03	100	109819	117843	7.306	109035	8.078	109184	7.931
t1d100.04	100	109252	117639	7.677	108417	8.506	108675	8.248
t1d100.05	100	108859	117538	7.973	108094	8.737	108447	8.383
t1d100.06	100	108201	117057	8.185	107786	8.601	107609	8.779
t1d100.07	100	108803	117118	7.642	108276	8.166	108405	8.037
t1d100.08	100	107480	115756	7.700	106746	8.440	107010	8.173
t1d100.09	100	108549	116527	7.350	107720	8.176	107776	8.120
t1d100.10	100	108771	117518	8.042	108222	8.590	108336	8.475

Instances RandomAI:

These instances are generated from a uniform distribution in the range $[0, 100]$. These problems were originally generated from a [0, 25000] uniform distribution [58] and modified afterwards, sampling from a significatively narrow range ([0,100]) to make them harder to solve. The size of these instances are 100, 150, and 200. For the experiments, we use the instances with size 100 as the larger instances are too difficult to be solved using IP.

5.5.2 Computational results

We summarize the computational results in Table 5.1 and Table 5.2. Table 5.1 shows experiments on special instances that can be solved in reasonable time using IP, and Table 5.2 shows experiments on the much harder RandomAI instances. In Table 5.1, we can see that using the heuristic (H), and using the heuristic along with node selection (H+NS), both lead to a decline in the number of LPs (NLPS) solved. However, for some instances we can see an increase in the CPU time (in seconds) which may be due to the additional work at each node. Table 5.2 shows results for the much harder instances for which we report the

lower bounds obtained. The heuristic can be used to find solutions which are substantially close to the best known solutions [69] thereby making it possible to prune large parts of the tree quickly.

5.6 Summary

The Linear Ordering Problem is a classic combinatorial optimization problem with applications to numerous problems of practical importance. A standard method to solve such problems optimally is using Integer Programming and branch-and-bound. The computation time, however, grows rapidly with the size of the problem instance. In this paper, we propose a new MIP-based primal heuristic that generates good feasible solutions from the partial LP solution at each node of the branch-and-bound tree. We also devise a new node selection strategy based on the heuristic solution. Preliminary experimental results show that the approach is promising. The solutions obtained using the heuristic are substantially close to the optimal and provide a good lower bound for the branch-and-bound algorithm. The number of nodes that need to be processed also show a decline when the heuristic is used.

Chapter 6

Discussion

In this thesis, our main contribution includes hybrid primal heuristics that combine both heuristic and exact methods. In particular, we propose a supervised large neighborhood search heuristic for the general mixed integer programming problem as well as a MIP-based heuristic for the linear ordering problem. This chapter outlines our results and discusses the limitations and scope of future work for the ideas presented in the thesis.

6.1 Supervised neighborhood selection

For our preliminary work on supervised neighborhood selection, we focus, particularly on the variable fixing neighborhood. We design the framework to predict the set of variables to be fixed in an ideal search neighborhood. We evaluate the effectiveness and generality of our large neighborhood search heuristic through extensive experimentation on a wide selection of problem instances. The experiments clearly demonstrate that our supervised learning approach to neighborhood selection is promising. Our heuristic is not only more likely to find an improving solution but also improves the solver performance over multiple metrics compared to the existing state-of-the-art heuristics. Here, we discuss its limitations and possible directions for future work.

6.1.1 Limitations

A key limitation of our heuristic is the computation time required by the neighborhood selection framework. The time complexity of the framework is linear in the number of non-zeros in the problem instance as compared to RINS, which has a time complexity linear in the number of variables. Moreover, the graph attention model adds a significant overhead to the computation time, especially when executed on a CPU. The heuristic is likely to be prohibitively expensive on larger or denser problem instances. We believe however, that the overall computation time can be considerably improved through a tighter integration with the MIP solver and through model execution on a GPU.

6.1.2 Future work

In the course of our preliminary work, we identified many promising avenues for further research.

Alternate and hybrid neighborhoods

In our exploratory study, we focus specifically on the variable fixing neighborhood. However, a straightforward extension of our work is to incorporate alternate neighborhoods proposed in the literature. Possible approaches include adding a global cut similar to local branching [32], or modifying the objective function coefficients like in proximity search [35]. Hybrid approaches combining the different techniques is another promising direction.

Transfer learning

Transfer learning is a compelling approach where an ML model trained on one high level task can be easily reused for another related but distinct task. It would be interesting to explore this idea in the context of mixed integer programming where we have several high level problems to be tackled such as branching strategies, node selection strategies, and cutting plane selection. It would also help justify the high computation cost involved in sophisticated machine learning models such as graph convolutional networks.

Interpretability

A common drawback of neural network models is their lack of interpretability in terms of the logic behind their predictions. Recent visualization techniques in the field of machine learning aim to alleviate this problem and try to explain the decisions made by the neural network models. Analyzing the learned graph network models may provide key insights into what information is most crucial when defining a search neighborhood. This might help in discovering new simpler heuristics reducing the time and memory overhead of our graph attention model. Another possibility may be the classification of problem instances based on the information most relevant to solve them.

Standardized open datasets

Lastly, areas such as computer vision have developed immensely due to standardized open datasets that help reduce the barrier of entry into the field. Given the difficulty in obtaining data for combinatorial optimization problems, a similar approach can lead to a more focused effort in this area. It would also provide an opportunity for a fairer comparison between different approaches. Important benefits that can come out of this exercise are better modeling techniques for MIP and more sophisticated features engineered to tackle the problem of generalizability.

6.2 Primal heuristic for the linear ordering problem

Another work, we present in this thesis, targets the linear ordering problem. We propose an analytical MIP-based primal heuristic for the linear ordering problem. The heuristic is based on a decomposition technique that subdivides the LOP instance into smaller instances that can be quickly solved using a MIP solver. The solution for the original problem is then constructed by concatenating the partial solutions. Our experiments demonstrate that the heuristic is fast and can be used as a primal heuristic in the branch-and-bound algorithm. The heuristic has the potential to be applied to very large scale instances as it uses a decomposition based approach.

6.2.1 Limitations

A limitation of our MIP-based heuristic for LOP is that the size of the instance that can be efficiently solved using a MIP solver is not clear. The actual solving time would depend on the type of instance and the particular instance at hand. The method seems to work well in the case of some real-world instances and random LOP instances. However, the heuristic might suffer greatly in the case of instances that are more symmetric in structure.

6.2.2 Future work

In the context of our work on the linear ordering problem, we identify two key future directions based on machine learning based approaches.

ML-based approaches to LOP

Many ML-based approaches have been proposed for the traveling salesman problem, specifically, the euclidean version. Typically, these methods rely on the approximability of the euclidean TSP. Even though the structure of the problem input is the same in the case of TSP and LOP, it would be interesting to see how similar methods fare on the linear ordering problem.

Decomposition based heuristics

Most of the proposed ML-based heuristics approach the problem targeting the local structure, i.e. through greedily constructing the solution or directly predicting it. An alternate approach is to use an ML-based decomposition and fall back on MIP to solve partial problems optimally.

Chapter 7

Conclusion

Our key contribution, in this thesis, includes the design of hybrid primal heuristics that combine both heuristic and exact methods. In particular, we propose a supervised neighborhood selection framework and incorporate it within a large neighborhood search heuristic for the general mixed integer program. Our framework uses data collected over a diverse set of problem instances, to devise a dynamic strategy for defining a variable fixing neighborhood. The heuristic searches for an improving solution by solving an auxiliary problem, a version of the original MIP restricted to our dynamically defined neighborhood. We perform extensive experimentation in order to evaluate our neighborhood selection framework and the resulting large neighborhood search heuristic. The experiments demonstrate that our supervised learning based approach to neighborhood selection is promising. Moreover, our heuristic also finds an improving solution more often than existing state-of-the-art heuristics and performs at par or better on many key metrics used for evaluation.

We also propose an analytical MIP-based primal heuristic for the linear ordering problem. The key idea is to recursively decompose the LOP problem instance into smaller ones until each instance is small enough to be quickly solved to optimality using a MIP solver. The heuristic then constructs the solution to the original LOP instance by concatenating the partial solutions. Although our heuristic is fast and can achieve good solutions, the solutions are not quite comparable to state-of-the-art methods such as memetic algorithm.

In the course of our work, we also identified many avenues for future work in integrating machine learning with mixed integer programming. Future work could be focused towards answering the following questions: to what extent can machine learning support problems in the context of MIP, what information is most critical when solving a MIP, and what are the characteristic features of MIP that make it difficult to solve.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] T. Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, jul 2009.

[3] T. Achterberg and T. Berthold. Improving the feasibility pump. *Discrete Optimization*, 4(1):77–86, 2007.

[4] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005.

[5] T. Achterberg and R. Wunderling. *Mixed Integer Programming: Analyzing 12 Years of Progress*, pages 449–481. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[6] R. Agrawal., E. Iranmanesh., and R. Krishnamurti. Primal heuristic for the linear ordering problem. In *Proceedings of the 8th International Conference on Operations Research and Enterprise Systems - Volume 1: ICORES*, pages 151–156. INSTICC, SciTePress, 2019.

[7] R. K. Ahuja, Ö. Ergun, J. B. Orlin, and A. P. Punnen. A Survey of Very Large-scale Neighborhood Search Techniques. *Discrete Appl. Math.*, 123(1-3):75–102, nov 2002.

[8] A. M. Alvarez, Q. Louveaux, and L. Wehenkel. A Machine Learning-Based Approximation of Strong Branching. *INFORMS J. on Computing*, 29(1):185–195, feb 2017.

[9] C. E. Andrade, S. Ahmed, G. L. Nemhauser, and Y. Shao. A hybrid primal heuristic for finding feasible solutions to mixed integer programs. *European Journal of Operational Research*, 263(1):62–71, 2017.

[10] E. Balas and C. H. Martin. Pivot and Complement-A Heuristic for 0-1 Programming. *Management Science*, 26(1):86–96, 1980.

[11] E. Balas, S. Schmieta, and C. Wallace. Pivot and shift—a mixed integer programming heuristic. *Discrete Optimization*, 1(1):3–12, 2004.

[12] M. Balcan, T. Dick, T. Sandholm, and E. Vitercik. Learning to Branch. *arXiv e-prints*, page arXiv:1803.10150, mar 2018.

[13] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural Combinatorial Optimization with Reinforcement Learning. *arXiv e-prints*, page arXiv:1611.09940, nov 2016.

[14] Y. Bengio, A. Lodi, and A. Prouvost. Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon. *arXiv e-prints*, page arXiv:1811.06128, nov 2018.

[15] L. Bertacco, M. Fischetti, and A. Lodi. A feasibility pump heuristic for general mixed-integer problems. *Discrete Optimization*, 4(1):63–76, 2007.

[16] T. Berthold. Primal Heuristics for Mixed Integer Programs. Master's thesis, Technische Universität Berlin, 2006.

[17] T. Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6):611–614, 2013.

[18] T. Berthold. RENS. *Mathematical Programming Computation*, 6(1):33–54, 2014.

[19] T. Berthold, G. Hendel, and T. Koch. From feasibility to improvement to proof: three phases of solving mixed-integer programs. *Optimization Methods and Software*, 33(3):499–517, 2017.

[20] E. R. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling. MIP: Theory and Practice — Closing the Gap. In M. J. D. Powell and S. Scholtes, editors, *System Modelling and Optimization*, pages 19–49, Boston, MA, 2000. Springer US.

[21] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM computing surveys (CSUR)*, 35(3):268–308, 2003.

[22] J. Ceberio, A. Mendiburu, and J. A. Lozano. The linear ordering problem revisited. *European Journal of Operational Research*, 241(3):686–696, mar 2015.

[23] T. Christof and G. Reinelt. Algorithmic Aspects of Using Small Instance Relaxations in Parallel Branch-and-Cut. *Algorithmica*, 30(4):597–629, jan 2001.

[24] G. Cornuéjols, D. Naddef, and W. R. Pulleyblank. Halin graphs and the travelling salesman problem. *Mathematical Programming*, 26(3):287–294, Oct 1983.

[25] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song. Learning Combinatorial Optimization Algorithms over Graphs. *arXiv e-prints*, page arXiv:1704.01665, apr 2017.

[26] E. Danna, E. Rothberg, and C. L. Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming*, 102(1):71–90, jan 2005.

[27] S. S. Dey and M. Molinaro. Theoretical challenges towards cutting-plane selection. *Mathematical Programming*, 170(1):237–266, 2018.

[28] G. Di Liberto, S. Kadioglu, K. Leo, and Y. Malitsky. DASH: Dynamic Approach for Switching Heuristics. *arXiv e-prints*, page arXiv:1307.4689, jul 2013.

[29] B. Dilkina, E. B. Khalil, and G. L. Nemhauser. Comments on: On learning and branching: a survey. *TOP*, 25(2):242–246, jul 2017.

[30] J. 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. *arXiv e-prints*, page arXiv:1906.09575, jun 2019.

[31] M. Fischetti, F. Glover, and A. Lodi. The feasibility pump. *Mathematical Programming*, 104(1):91–104, sep 2005.

[32] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98(1):23–47, sep 2003.

[33] M. Fischetti and A. Lodi. *Heuristics in Mixed Integer Programming*. American Cancer Society, 2011.

[34] M. Fischetti, A. Lodi, and D. Salvagnin. *Just MIP it!*, pages 39–70. Springer US, Boston, MA, 2010.

[35] M. Fischetti and M. Monaci. Proximity search for 0-1 mixed-integer convex programming. *Journal of Heuristics*, 20(6):709–731, dec 2014.

[36] G. Gamrath, T. Berthold, S. Heinz, and M. Winkler. Structure-Based Primal Heuristics for Mixed Integer Programming. In K. Fujisawa, Y. Shinano, and H. Waki, editors, *Optimization in the Real World*, pages 37–53, Tokyo, 2016. Springer Japan.

[37] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[38] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact Combinatorial Optimization with Graph Convolutional Neural Networks. *arXiv e-prints*, page arXiv:1906.01629, jun 2019.

[39] A. Georges, A. Gleixner, G. Gojic, R. L. Gottwald, D. Haley, G. Hendel, and B. Matejczyk. Feature-Based Algorithm Selection for Mixed Integer Programming. Technical Report 18-17, Zuse Institute Berlin (ZIB), Takustr. 7, 14195 Berlin, 2018.

[40] S. Ghosh. DINS, a MIP Improvement Heuristic. In M. Fischetti and D. P. Williamson, editors, *Integer Programming and Combinatorial Optimization*, pages 310–323, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[41] A. Gleixner, T. Achterberg, P. Christophel, M. Lübbecke, T. K. Ralphs, G. Hendel, G. Gamrath, M. Bastubbe, T. Berthold, K. Jarck, and Others. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. Technical report, Zuse Institute Berlin (ZIB), 2019.

[42] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano,

J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 6.0. Technical report, Optimization Online, jul 2018.

[43] M. Guzelsoy, G. Nemhauser, and M. Savelsbergh. Restrict-and-relax search for 0-1 mixed-integer programs. *EURO Journal on Computational Optimization*, 1(1):201–218, may 2013.

[44] S. Hanafi, J. Lazić, and N. Mladenović. Variable Neighbourhood Pump Heuristic for 0-1 Mixed Integer Programming Feasibility. *Electronic Notes in Discrete Mathematics*, 36:759–766, 2010.

[45] H. He, H. Daume III, and J. M. Eisner. Learning to Search in Branch and Bound Algorithms. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3293–3301. Curran Associates, Inc., 2014.

[46] G. Hendel. New Rounding and Propagation Heuristics for Mixed Integer Programming, 2011.

[47] G. Hendel. Adaptive Large Neighborhood Search for Mixed Integer Programming. Technical Report 18-60, Zuse Institute Berlin (ZIB), Takustr. 7, 14195 Berlin, 2018.

[48] J. J. Hopfield and D. W. Tank. "Neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52(3):141–152, jul 1985.

[49] IBM. CPLEX User's Manual, Version 12.9.0. 1987 - 2019.

[50] E. Iranmanesh. *Algorithms for Problems in Voting and Scheduling*. PhD thesis, Simon Fraser University, 2016.

[51] E. Iranmanesh. and R. Krishnamurti. Mixed integer program heuristic for linear ordering problem. In *Proceedings of 5th the International Conference on Operations Research and Enterprise Systems - Volume 1: ICORES*, pages 152–156. INSTICC, SciTePress, 2016.

[52] R. M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[53] E. B. Khalil, P. L. Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to Branch in Mixed Integer Programming. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 724–731. AAAI Press, 2016.

[54] E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao. Learning to Run Heuristics in Tree Search. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, pages 659–666. AAAI Press, 2017.

[55] F. Kılınç Karzan, G. L. Nemhauser, and M. W. P. Savelsbergh. Information-based branching schemes for binary linear mixed integer problems. *Mathematical Programming Computation*, 1(4):249–293, dec 2009.

[56] W. Kool, H. van Hoof, and M. Welling. Attention, Learn to Solve Routing Problems! *arXiv e-prints*, page arXiv:1803.08475, mar 2018.

[57] M. Kruber, M. E. Lübbecke, and A. Parmentier. Learning when to use a decomposition. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 202–210. Springer, 2017.

[58] M. Laguna, R. Marti, and V. Campos. Intensification and diversification with elite tabu search solutions for the linear ordering problem. *Computers & Operations Research*, 26(12):1217–1230, 1999.

[59] H. Lemos, M. Prates, P. Avelar, and L. Lamb. Graph Colouring Meets Deep Learning: Effective Graph Neural Network Models for Combinatorial Problems. *arXiv e-prints*, page arXiv:1903.04598, mar 2019.

[60] Z. Li, Q. Chen, and V. Koltun. Combinatorial Optimization with Graph Convolutional Networks and Guided Tree Search. *arXiv e-prints*, page arXiv:1810.10659, oct 2018.

[61] S. Lin and B. W. Kernighan. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, 21(2):498–516, apr 1973.

[62] A. Lodi. *The Heuristic (Dark) Side of MIP Solvers*, pages 273–284. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[63] A. Lodi, L. Mossina, and E. Rachelson. Learning to Handle Parameter Perturbations in Combinatorial Optimization: an Application to Facility Location. *arXiv e-prints*, page arXiv:1907.05765, jul 2019.

[64] A. Lodi and A. Tramontani. Performance Variability in Mixed-Integer Programming. In *Theory Driven by Influential Applications*. INFORMS, 2013.

[65] A. Lodi and G. Zarpellon. On learning and branching: a survey. *TOP*, 25(2):207–236, jul 2017.

[66] A. Lokketangen and F. Glover. Solving zero-one mixed integer programming problems using tabu search. *European Journal of Operational Research*, 106(2):624–658, 1998.

[67] 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*, volume 9725, pages 301 – 307, 2016.

[68] R. Martí and G. Reinelt. *The Linear Ordering Problem: Exact and Heuristic Methods in Combinatorial Optimization*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[69] R. Martí, G. Reinelt, and A. Duarte. Optsicom Project. *http: www.optsicom.es/lolib*, 2009.

[70] J. E. Mitchell and B. Borchers. *Solving Linear Ordering Problems with a Combined Interior Point/Simplex Cutting Plane Algorithm*, pages 349–366. Springer US, Boston, MA, 2000.

[71] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[72] V. T. Paschos. *Applications of combinatorial optimization*. John Wiley & Sons, 2013.

[73] E. Rothberg. An Evolutionary Algorithm for Polishing Mixed Integer Programming Solutions. *INFORMS Journal on Computing*, 19:534–541, 2007.

[74] T. Schiavinotto and T. Stützle. The linear ordering problem: Instances, search space analysis and algorithms. *Journal of Mathematical Modelling and Algorithms*, 3(4):367–402, jan 2005.

[75] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill. Learning a SAT Solver from Single-Bit Supervision. *arXiv e-prints*, page arXiv:1802.03685, feb 2018.

[76] J. Song, R. Lanka, A. Zhao, A. Bhatnagar, Y. Yue, and M. Ono. Learning to Search via Retrospective Imitation. *arXiv e-prints*, page arXiv:1804.00846, apr 2018.

[77] Y. Tang, S. Agrawal, and Y. Faenza. Reinforcement Learning for Integer Programming: Learning to Cut. *arXiv e-prints*, page arXiv:1906.04859, jun 2019.

[78] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph Attention Networks. *arXiv e-prints*, page arXiv:1710.10903, oct 2017.

[79] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer Networks. *arXiv e-prints*, page arXiv:1506.03134, jun 2015.

[80] D. P. Williamson and D. B. Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

[81] L. A. Wolsey. Integer programming. *IIE Transactions*, 32(273-285):2–58, 2000.

[82] M. Zhu. Recall, precision and average precision. *Department of Statistics and Actuarial Science, University of Waterloo, Waterloo*, 2:30, 2004.

Appendix A

Description of Features

For our graph attention model, we use many features that have been quite commonly used in the recent work in this area. The references for each feature are inline.

Feature	Description	Count	Reference
Basic Features (12)			
Type	(binary, integer, implicit integer, continuous)	4	[38, 30]
Coefficient, normalized	(raw, positive only, negative only)	3	[53, 38, 30]
Number of constraints	Number of nonzero coefficients for variable	1	[53, 30]
Has lower (upper) bound	Lower (upper) bound indicator	2	[38]
Global lower (upper) bound, standardized	Value of global lower (upper) bound	2	[30]
LP Features (16)			
Solution value, standardized	Value in LP-relaxation solution	1	[38, 30]
Slack and ceil distances	$\min\{\hat{x} - \lfloor\hat{x}\rfloor, \lceil\hat{x}\rceil - \hat{x}\}$ and $\lceil\hat{x}\rceil - \hat{x}$	2	[53, 30]
Basis status	(lower, basic, upper)	3	[38]
Reduced costs, standardized	Value of reduced cost	1	[38, 30]
Pseudocosts, standardized	Upwards, downwards, sum, product, ratio	5	[53, 30]
May round up (down)	Variable may round up (down)?	2	[54]
Number of up (down) locks	Number of locks	2	[54, 30]
Aggregated/Structure Features (44)			
Stats for constraint degrees	(mean, std. dev., min, max)	4	[53, 30]
Stats for positive (negative) constraint coefficients	(count, mean, std. dev., min, max)	10	[53, 30]
Ratios of constraint coefficients to RHS, standardized	(min, max)	4	[53, 30]
One-to-all coefficient ratios	(min, max)	8	[53]
Stats for active constraint coefficients, standardized. Three weighting schemes: unit weight, dual cost, inverse sum of coefficients	(count, sum, mean, std. dev., min, max)	18	[53, 30]
History/Branch-and-bound Features (8)			
Age	Age of column in LP	1	[38]
Fraction up (down) infeasibility	Fraction of times branch is infeasible	2	[53]
Local lower (upper) bound, standardized	Value of local lower (upper) bound	2	
Incumbent value, standardized	Value in incumbent solution	1	[38]
Average incumbent value, standardized	Value of average incumbent solution	1	[38]
Solution is at incumbent	Is solution value and incumbent value equal?	1	

Table A.1: Description of Variable Features ($\mathbf{V}$)

Feature	Description	Count	Reference
Basic Features (12)			
Objective cosine similarity	Cosine similarity with objective function	1	[38]
RHS, standardized	Value of RHS in constraint	1	[38, 30]
Number of nonzeros (positive, negative)	Number of nonzeros in constraint	3	[30]
Constraint types	(precedence, and, knapsack, linear, logicor, setppc, varbound)	7	[30]
LP Features (4)			
Dual solution, standardized	Dual solution for the constraint	1	[38, 30]
Basis Status	(lower, basic, upper)	3	[30]
Aggregated Features (7)			
Sum norm of coefficients	(absolute, positive only, negative only)	3	[30]
Stats on variable coefficients	(mean, std. dev., min, max)	4	[30]
History/Branch-and-bound Features (1)			
Age	Age of row in LP	1	[38]

Table A.2: Description of Constraint Features (**C**)

Feature scaling is typically used to stabilize the training of machine learning models. While scaling features is important, it also causes loss of information. This becomes especially tricky in our case due to the high variability in problem structure and feature scales across a diverse set of instances. We address this problem by choosing the most natural way to scale the features. For example, the objective function is scaled by dividing by the L2 norm. Similarly, the number of constraints is scaled by dividing by the total number of constraints. This works well in practice as presolving can detect and help remove the redundancies in a problem instance. When there is no clear and intuitive way of scaling a feature, we use query based standardization similar to [53]. The features that are standardized are noted inline.

Feature	Description	Count	Reference
Basic Features (1)			
Variable coefficient in constraint, normalized	Coefficient of the variable in constraint	1	[38, 30]

Table A.3: Description of Edge Features (**E**)

Appendix B

Exhaustive Results for MIP Instances

Table B.1: Exhaustive Results for MIP Instances - Primal Gap, Primal Dual Gap, Primal Integral, Time to Best Solution

Instance	Primal Gap		Primal Dual Gap		Primal Integral		Time to Best Sol.	
	w/o SLNS	w SLNS	w/o SLNS	w SLNS	w/o SLNS	w SLNS	w/o SLNS	w SLNS
30n20b8.mps	0.00	0.00	0.00	0.00	129.74	**103.39**	396.74	**351.58**
50v-10.mps	0.91	**0.49**	2.95	**2.43**	32.48	**22.45**	1201.04	**759.07**
CMS750_4.mps	18.11	**16.09**	18.76	**16.76**	777.68	**736.06**	**1646.08**	2908.98
academictimetablesmall.mps	100.00	100.00	100.00	100.00	3600.01	3600.01	**1279.80**	1575.88
air05.mps	0.00	0.00	0.00	0.00	0.85	**0.84**	**30.11**	30.26
app1-1.mps	0.00	0.00	0.00	0.00	**0.63**	0.64	9.27	**9.09**
app1-2.mps	0.00	0.00	0.00	0.00	3.65	**3.60**	819.30	**792.57**
assign1-5-8.mps	0.00	0.00	6.00	**5.74**	0.28	**0.27**	8.62	**7.49**
atlanta-ip.mps	2.98	**2.16**	9.60	**8.55**	157.30	**137.26**	**2639.46**	2654.64
b1c1s1.mps	2.55	**1.01**	14.37	**12.53**	159.42	**121.55**	**1898.52**	2183.28
beasleyC3.mps	0.00	0.00	0.28	**0.18**	2.63	**2.04**	311.80	**159.83**
binkar10_1.mps	0.00	0.00	0.00	0.00	0.06	**0.04**	42.43	**41.49**
blp-ar98.mps	1.95	**1.63**	2.62	**2.30**	113.08	**100.73**	**2420.60**	2722.73
blp-ic98.mps	**5.41**	6.03	**6.61**	7.18	**204.06**	236.75	2674.26	**1781.34**
bnatt400.mps	0.00	0.00	0.00	0.00	0.00	0.00	355.74	**330.42**
bppc4-08.mps	0.00	0.00	1.89	1.89	**1.00**	1.39	565.72	**466.13**
brazil3.mps	1.49	**0.00**	**5.95**	6.07	354.17	**331.29**	1243.68	**1115.71**
chromaticindex1024-7.mps	0.00	0.00	25.00	25.00	0.00	0.00	**2793.19**	2895.45
chromaticindex512-7.mps	0.00	0.00	**14.33**	19.54	0.00	0.00	**847.31**	870.11
cmflsp50-24-8-8.mps	2.54	**1.48**	3.61	**2.51**	150.50	**145.73**	**2732.62**	3033.62
cod105.mps	0.00	0.00	26.07	**26.01**	0.00	0.00	146.15	**145.24**
comp07-2idx.mps	93.62	**93.14**	93.62	**93.14**	3454.94	**3440.18**	**2930.92**	3309.59
comp21-2idx.mps	52.49	**49.24**	71.36	**68.81**	2131.10	**2084.02**	2983.86	**2606.02**
cost266-UUE.mps	0.00	0.00	6.08	**4.89**	11.25	**5.82**	1375.92	**985.99**
csched007.mps	1.06	**0.33**	10.63	**9.54**	119.91	**57.82**	3002.65	**1663.21**
csched008.mps	0.00	0.00	0.00	0.00	8.29	**4.94**	909.68	**651.25**
cvs16r128-89.mps	1.65	**0.62**	20.39	**19.37**	100.83	**65.81**	**1363.93**	1470.43
dano3_3.mps	0.00	0.00	0.00	0.00	0.17	**0.15**	105.07	**100.41**
dano3_5.mps	0.01	**0.00**	0.02	**0.00**	0.83	**0.60**	**267.08**	331.46
drayage-100-23.mps	0.00	0.00	0.00	0.00	0.16	**0.14**	14.01	**13.15**
drayage-25-23.mps	0.00	0.00	0.07	**0.06**	1.65	**1.58**	**346.27**	547.35
dws008-01.mps	27.95	**20.56**	61.24	**55.92**	1010.91	**848.08**	**1000.46**	2696.05
eil33-2.mps	0.00	0.00	0.00	0.00	**5.51**	5.57	**131.61**	132.55
fast0507.mps	0.00	0.00	0.00	0.00	0.91	**0.90**	131.02	**125.62**
fastxgemm-n2r6s0t2.mps	0.00	0.00	87.88	**87.26**	29.89	**27.54**	264.87	**100.05**
fiball.mps	1.70	**0.00**	1.70	**0.00**	61.22	**24.61**	**1382.52**	1704.01
gen-ip002.mps	0.03	**0.00**	0.59	**0.48**	0.97	**0.29**	**545.54**	2114.26
gen-ip054.mps	0.12	**0.11**	0.89	**0.82**	6.69	**3.54**	**546.65**	2347.02
germanrr.mps	**1.18**	1.44	**1.74**	2.00	**69.92**	72.87	**2259.80**	2600.37
glass-sc.mps	0.00	0.00	14.78	**14.47**	0.00	0.00	161.79	**147.34**
glass4.mps	23.74	**15.08**	46.11	**31.85**	955.72	**730.12**	**1653.78**	2444.37
gmu-35-40.mps	0.02	0.02	0.03	0.03	1.45	**1.03**	1977.37	**1426.71**
gmu-35-50.mps	0.05	**0.04**	0.05	**0.04**	2.46	**1.78**	**1406.72**	3194.51
graph20-20-1rand.mps	0.00	0.00	65.77	**65.68**	33.88	**31.95**	225.12	**211.26**
graphdraw-domain.mps	1.07	**0.01**	24.40	**18.82**	73.97	**18.47**	**1368.36**	1783.11
h80x6320d.mps	0.00	0.00	0.00	0.00	0.00	0.00	191.43	**164.36**
ic97_potential.mps	0.38	**0.08**	2.00	**1.60**	28.32	**11.03**	**1703.92**	1729.84
icir97_tension.mps	0.57	**0.06**	0.83	**0.32**	21.51	**7.10**	**2210.62**	2366.88

Continued on next page ...

Table B.1 – continued from previous page

Instance	Primal Gap		Primal Dual Gap		Primal Integral		Time to Best Sol.	
	w/o SLNS	w SLNS	w/o SLNS	w SLNS	w/o SLNS	w SLNS	w/o SLNS	w SLNS
irp.mps	0.00	0.00	0.00	0.00	0.00	0.00	**18.89**	19.63
istanbul-no-cutoff.mps	0.00	0.00	0.00	0.00	13.76	**13.00**	186.00	**175.36**
lectsched-5-obj.mps	37.06	**29.89**	60.74	**56.36**	1567.51	**1301.92**	1386.23	**1143.91**
leo1.mps	2.16	**1.41**	3.41	**2.54**	92.99	**74.13**	**1750.67**	2570.31
leo2.mps	4.69	**2.63**	6.46	**4.35**	212.46	**157.11**	**2213.82**	3001.37
lotsize.mps	0.54	0.54	1.51	**1.50**	156.43	**147.24**	2202.57	**2119.31**
mad.mps	44.92	**27.31**	100.00	100.00	1944.45	**1581.00**	**2286.91**	2341.01
markshare2.mps	93.59	**93.21**	100.00	100.00	**3407.31**	3408.20	**2364.96**	2564.01
markshare_4_0.mps	0.00	0.00	**74.11**	100.00	53.26	**19.70**	1364.26	**640.57**
mas74.mps	0.39	**0.00**	5.69	**4.24**	11.38	**1.18**	684.63	**110.29**
mas76.mps	0.00	0.00	0.44	**0.00**	0.04	0.04	3.71	**2.61**
mc11.mps	0.04	**0.00**	1.61	**1.55**	2.52	**1.30**	**629.78**	726.93
mcsched.mps	0.00	0.00	0.00	0.00	0.66	**0.65**	85.96	**85.77**
mik-250-20-75-4.mps	0.00	0.00	0.00	0.00	0.00	0.00	12.67	**9.55**
milo-v12-6-r2-40-1.mps	0.83	**0.27**	18.53	**17.75**	109.87	**74.47**	**1450.19**	1960.77
momentum1.mps	24.14	**12.75**	32.48	**21.68**	1502.97	**1481.12**	**2543.65**	3277.19
mushroom-best.mps	0.00	0.00	68.91	**62.00**	13.01	**12.33**	116.98	**105.20**
mzzv11.mps	0.00	0.00	0.00	0.00	8.19	**7.91**	257.81	**235.45**
mzzv42z.mps	0.00	0.00	0.00	0.00	**0.95**	0.96	**154.74**	155.49
n2seq36q.mps	0.00	0.00	0.38	0.38	21.93	**19.70**	221.83	**206.54**
n3div36.mps	0.09	0.09	4.39	**4.32**	30.48	**27.96**	**709.20**	737.17
n5-3.mps	0.00	0.00	0.00	0.00	6.52	**6.31**	31.95	**30.85**
neos-1171448.mps	0.00	0.00	0.00	0.00	0.16	**0.12**	56.32	**51.44**
neos-1171737.mps	2.56	**1.18**	2.56	**1.18**	90.53	**46.57**	**203.39**	1139.41
neos-1445765.mps	0.00	0.00	0.00	0.00	0.18	0.18	77.05	**73.16**
neos-1456979.mps	0.11	0.11	7.58	**7.48**	37.77	**25.79**	1953.30	**1095.66**
neos-1582420.mps	0.00	0.00	0.00	0.00	0.22	**0.21**	25.82	**24.35**
neos-2657525-crna.mps	51.85	**31.44**	100.00	100.00	2103.54	**249.72**	**218.63**	544.93
neos-2746589-doon.mps	0.23	**0.15**	0.55	**0.46**	**81.02**	89.12	2015.22	**1656.38**
neos-3004026-krka.mps	0.00	0.00	0.00	0.00	0.00	0.00	**316.32**	327.72
neos-3024952-loue.mps	31.13	**16.00**	33.83	**19.30**	1253.30	**999.17**	**2641.96**	3243.38
neos-3046615-murg.mps	3.46	**0.02**	73.99	**69.28**	136.25	**6.43**	**715.60**	962.58
neos-3083819-nubu.mps	0.00	0.00	0.00	0.00	0.03	0.03	34.61	**25.60**
neos-3216931-puriri.mps	54.70	54.70	62.01	**61.98**	1968.96	1968.96	**1507.41**	1516.65
neos-3381206-awhea.mps	0.00	0.00	0.00	0.00	**0.03**	0.04	**5.95**	6.21
neos-3627168-kasai.mps	0.07	0.07	0.51	**0.49**	3.03	**2.87**	**1042.06**	1226.01
neos-3656078-kumeu.mps	6.01	**1.02**	11.95	**7.44**	425.27	**359.20**	**1917.42**	3194.27
neos-4300652-rahue.mps	12.16	12.16	89.77	**89.63**	**838.35**	840.45	**1920.98**	1944.64
neos-4338804-snowy.mps	0.77	**0.51**	2.39	**2.14**	72.08	**53.99**	2483.62	**2010.25**
neos-4387871-tavua.mps	4.09	**1.54**	20.56	**18.70**	185.01	**87.00**	**934.05**	1927.83
neos-4722843-widden.mps	0.00	0.00	**4.54**	5.24	126.36	**123.49**	3517.60	**3367.53**
neos-4738912-atrato.mps	0.00	0.00	0.00	0.00	1.42	**0.78**	974.50	**421.73**
neos-4763324-toguru.mps	1.30	**1.15**	26.81	**26.15**	219.12	**211.80**	**2102.39**	2367.37
neos-4954672-berkel.mps	1.02	**0.49**	14.33	**13.66**	46.25	**30.93**	914.80	**517.21**
neos-5107597-kakapo.mps	50.96	**22.13**	90.66	**82.97**	2458.10	**1742.87**	**3367.00**	3404.52
neos-5188808-nattai.mps	0.00	0.00	0.00	0.00	45.94	**41.96**	1199.15	**1068.10**
neos-5195221-niemur.mps	0.14	**0.01**	67.77	**49.84**	600.84	**346.69**	2170.57	**1855.72**
neos-662469.mps	8.62	**0.07**	8.63	**0.07**	330.07	**281.61**	**1841.57**	2325.71
neos-873061.mps	**1.50**	2.04	**2.41**	2.94	**86.86**	91.77	3147.51	**3062.94**
neos-911970.mps	0.00	0.00	0.18	0.18	0.67	**0.44**	**1067.61**	1271.33
neos-933966.mps	0.00	0.00	0.00	0.00	3.00	**2.93**	**206.46**	208.43
neos-950242.mps	0.00	0.00	71.43	**71.21**	0.00	0.00	617.04	**569.15**
neos-957323.mps	0.00	0.00	0.00	0.00	0.00	0.00	**57.14**	57.24
neos-960392.mps	0.00	0.00	0.00	0.00	1.17	**1.07**	296.35	**257.67**
neos17.mps	0.00	0.00	0.00	0.00	0.76	**0.58**	42.25	**12.53**
neos5.mps	0.00	0.00	4.78	**3.22**	0.14	0.14	9.97	**8.78**
net12.mps	0.00	0.00	0.00	0.00	50.73	**42.34**	424.93	**388.09**
nexp-150-20-8-5.mps	0.86	**0.52**	1.35	**1.00**	46.74	**40.75**	**1424.11**	1913.89
ns1208400.mps	0.00	0.00	0.00	0.00	0.00	0.00	573.40	**464.26**
ns1830653.mps	0.91	**0.00**	6.14	**0.00**	72.78	**30.97**	488.73	**378.76**
ns1952667.mps	0.00	0.00	0.00	0.00	0.00	0.00	1651.08	**1297.10**
nu25-pr12.mps	0.00	0.00	0.00	0.00	0.00	0.00	6.72	**5.78**
nursesched-sprint02.mps	0.00	0.00	0.00	0.00	3.31	**2.78**	64.06	**54.64**
opm2-z10-s4.mps	**7.12**	10.39	**30.11**	33.35	**435.76**	472.26	**3070.28**	3096.47
p200x1188c.mps	0.00	0.00	0.00	0.00	0.00	0.00	**6.62**	6.96
peg-solitaire-a3.mps	0.00	0.00	0.00	0.00	0.00	0.00	**2482.96**	2829.42
pg.mps	0.00	0.00	0.00	0.00	0.01	0.01	19.03	**18.49**
pg5_34.mps	0.01	**0.00**	0.05	**0.00**	0.97	**0.23**	2253.57	**1411.25**
physicianshed6-2.mps	0.00	0.00	0.00	0.00	0.00	0.00	**117.78**	119.49
piperout-08.mps	0.00	0.00	0.00	0.00	99.14	**97.88**	269.44	**263.74**
piperout-27.mps	0.00	0.00	0.00	0.00	149.94	**146.89**	360.27	**352.27**
pk1.mps	0.00	0.00	0.00	0.00	**8.10**	8.58	**137.65**	416.65
radiationm18-12-05.mps	16.50	**9.21**	16.51	**9.21**	794.14	**568.34**	**1526.84**	2133.41
radiationm40-10-02.mps	53.68	**46.73**	53.68	**46.74**	1988.44	**1818.85**	**1756.72**	2874.20
rail01.mps	23.93	23.93	27.62	27.62	861.66	861.66	**3326.36**	3501.25
rail507.mps	0.00	0.00	0.00	0.00	1.18	**1.04**	94.70	**82.88**
ran14x18-disj-8.mps	0.51	**0.49**	4.15	**3.94**	20.51	**18.69**	**1116.52**	1749.06
reblock115.mps	0.05	**0.01**	0.48	**0.41**	21.65	**18.73**	1236.52	**976.09**
rmatr100-p10.mps	0.00	0.00	0.00	0.00	3.64	**3.18**	52.00	**45.95**
rmatr200-p5.mps	0.29	**0.14**	17.78	**16.97**	22.19	**7.11**	1872.88	**1529.66**
rocI-4-11.mps	0.00	0.00	0.00	0.00	19.85	**19.06**	81.96	**79.12**
rocII-5-11.mps	**2.98**	2.99	45.22	45.22	323.18	**305.69**	**2471.77**	2999.80
rococoB10-011000.mps	0.77	**0.04**	14.51	**13.38**	77.75	**52.85**	**1398.52**	1639.97
rococoC10-001000.mps	0.00	0.00	1.99	**1.65**	8.63	**5.03**	1078.79	**1017.29**
roll3000.mps	0.00	0.00	0.00	0.00	4.41	**2.32**	793.46	**377.70**

Continued on next page ...

Table B.1 – continued from previous page

Instance	Primal Gap		Primal Dual Gap		Primal Integral		Time to Best Sol.	
	w/o SLNS	w SLNS	w/o SLNS	w SLNS	w/o SLNS	w SLNS	w/o SLNS	w SLNS
satellites2-40.mps	**21.05**	57.89	**46.43**	71.43	**1621.78**	2084.22	3462.81	**2513.55**
satellites2-60-fs.mps	0.00	0.00	34.48	34.48	937.05	**857.89**	3350.81	**3076.20**
sct2.mps	0.00	0.00	0.04	**0.03**	0.75	**0.30**	674.37	**210.80**
seymour.mps	0.47	**0.19**	2.28	**1.96**	22.45	**11.67**	**1019.09**	1120.77
seymour1.mps	0.00	0.00	0.00	0.00	1.09	**1.08**	**135.43**	144.47
sing326.mps	0.26	**0.12**	0.43	**0.28**	26.19	**21.35**	3324.02	**3249.82**
sing44.mps	**0.07**	0.11	**0.19**	0.23	**9.62**	10.88	**2004.60**	3022.64
snp-02-004-104.mps	**0.95**	0.96	**0.96**	0.97	**79.13**	80.55	2606.80	**2065.82**
sp150x300d.mps	0.00	0.00	5.79	**4.35**	0.00	0.00	**0.99**	1.00
sp97ar.mps	2.16	**0.58**	2.78	**1.16**	102.08	**66.95**	**1969.08**	3033.82
sp98ar.mps	0.36	**0.22**	0.75	**0.59**	30.64	**29.31**	**2489.63**	2715.42
supportcase18.mps	4.00	4.00	5.63	5.63	147.74	**147.29**	644.03	**524.05**
supportcase26.mps	2.51	**0.40**	18.01	**15.18**	101.74	**44.47**	**846.40**	1060.63
supportcase33.mps	**0.00**	0.29	**0.00**	1.64	**49.25**	50.53	1753.48	**1017.86**
supportcase40.mps	0.70	**0.00**	3.22	**0.25**	26.02	**6.12**	**1499.09**	2145.31
supportcase42.mps	2.39	**2.37**	2.54	**2.53**	111.36	**107.70**	**923.38**	1239.90
swath1.mps	0.00	0.00	0.00	0.00	**0.34**	0.41	45.44	**45.15**
swath3.mps	0.24	**0.00**	6.81	**5.45**	6.39	**1.61**	**237.23**	347.23
tbfp-network.mps	0.00	0.00	0.00	0.00	122.56	**115.30**	1382.24	**1334.06**
thor50dday.mps	0.55	**0.54**	20.21	**20.20**	75.96	**66.64**	2886.05	**2787.95**
timtab1.mps	**0.24**	0.26	15.37	**10.89**	49.90	**22.91**	1376.06	**792.25**
tr12-30.mps	0.00	0.00	0.08	**0.07**	**0.18**	0.19	**577.08**	897.24
traininstance2.mps	9.10	**8.44**	100.00	100.00	379.56	**378.33**	1355.36	**1194.14**
traininstance6.mps	2.02	**1.32**	98.75	**98.40**	125.41	**89.94**	1525.86	**1478.90**
trento1.mps	2.08	**0.78**	2.17	**0.87**	189.64	**178.66**	**2649.32**	2797.51
uccase12.mps	0.00	0.00	0.00	0.00	0.00	0.00	**772.56**	981.64
uccase9.mps	**1.43**	1.64	**2.47**	2.68	110.83	**109.38**	**2257.27**	2865.60
uct-subprob.mps	0.38	**0.13**	7.24	**6.95**	32.86	**18.19**	**1112.00**	1123.40
unitcal_7.mps	0.00	0.00	0.00	0.00	0.00	0.00	216.92	**213.03**
wachplan.mps	0.00	0.00	11.11	11.11	0.00	0.00	26.37	**25.45**

Table B.2: Exhaustive Results for MIP Instances - Total Time, Total Nodes, Fraction Solved, Success Count

Instance	Total Time		Total Nodes		Fraction Solved		Success Count	
	w/o SLNS	w SLNS	w/o SLNS	w SLNS	w/o SLNS	w SLNS	RINS	SLNS
30n20b8.mps	405.82	**357.18**	430.36	**370.68**	1.00	1.00	0.00	**0.50**
50v-10.mps	3600.08	**3600.07**	**88157.10**	264664.90	0.00	0.00	0.00	**1.00**
CMS750_4.mps	**3600.22**	3600.42	**21994.67**	39784.93	0.00	0.00	0.20	**12.60**
academictimetablesmall.mps	3600.56	**3600.49**	**1444.34**	1511.53	0.00	0.00	0.00	**0.33**
air05.mps	**31.57**	31.73	259.17	259.17	1.00	1.00	0.00	0.00
app1-1.mps	9.56	**9.39**	13.60	13.60	1.00	1.00	0.00	0.00
app1-2.mps	1177.27	**1110.23**	841.47	**832.00**	1.00	1.00	0.25	0.25
assign1-5-8.mps	3600.13	**3600.09**	**851747.13**	1250871.21	0.00	0.00	**0.60**	0.40
atlanta-ip.mps	**3600.77**	3601.10	4025.66	**3785.42**	0.00	0.00	1.00	**2.00**
b1c1s1.mps	3600.17	**3600.16**	**43560.90**	71037.82	0.00	0.00	1.20	**13.60**
beasleyC3.mps	**1459.68**	1594.76	**45985.46**	60020.25	0.60	**0.80**	1.00	**2.20**
binkar10_1.mps	289.42	**258.19**	**53417.25**	54301.09	1.00	1.00	0.80	**2.80**
blp-ar98.mps	**3600.36**	3600.83	**14291.28**	30007.56	0.00	0.00	**3.40**	0.00
blp-ic98.mps	**3600.45**	3600.57	**24861.03**	37088.23	0.00	0.00	**2.00**	0.20
bnatt400.mps	451.57	**419.84**	5956.97	5956.97	1.00	1.00	0.00	0.00
bppc4-08.mps	3600.13	3600.13	**311043.16**	321163.86	0.00	0.00	0.00	**0.20**
brazil3.mps	2088.18	**2039.44**	**1217.44**	1361.16	0.60	0.60	0.00	**0.60**
chromaticindex1024-7.mps	3615.61	**3601.08**	160.87	**95.46**	0.00	0.00	0.00	0.00
chromaticindex512-7.mps	**3071.57**	3139.06	6310.79	**6110.49**	**0.40**	0.20	0.00	0.00
cmflsp50-24-8-8.mps	3600.38	**3600.27**	**33102.25**	38814.22	0.00	0.00	2.00	**12.00**
cod105.mps	3600.11	**3600.10**	**18118.91**	28285.73	0.00	0.00	0.00	0.00
comp07-2idx.mps	3600.39	**3600.28**	**2992.85**	4024.93	0.00	0.00	**1.40**	0.00
comp21-2idx.mps	**3600.18**	3600.30	**12695.29**	12743.41	0.00	0.00	5.00	**8.60**
cost266-UUE.mps	3600.11	3600.11	**44819.04**	113350.92	0.00	0.00	1.40	**5.60**
csched007.mps	3600.06	**3600.05**	**202982.55**	270726.97	0.00	0.00	2.60	**17.20**
csched008.mps	1894.00	**1812.16**	**113728.48**	125166.23	1.00	1.00	2.40	**4.60**
cvs16r128-89.mps	3600.13	**3600.10**	**17431.13**	30459.73	0.00	0.00	0.40	**2.40**
dano3_3.mps	112.08	**107.11**	12.16	12.16	1.00	1.00	0.00	0.00
dano3_5.mps	441.71	**380.49**	**233.08**	249.04	0.80	**1.00**	0.20	0.20
drayage-100-23.mps	14.40	**13.63**	6.23	6.23	1.00	1.00	0.00	0.00
drayage-25-23.mps	3600.22	**3518.82**	**141945.39**	253946.34	0.00	**0.20**	0.20	**1.60**
dws008-01.mps	3600.25	**3600.16**	**18846.24**	26898.38	0.00	0.00	**0.80**	0.00
eil33-2.mps	**216.89**	219.30	642.75	642.75	1.00	1.00	0.00	0.00
fast0507.mps	160.40	**154.68**	1098.18	1098.18	1.00	1.00	**0.20**	0.00
fastxgemm-n2r6s0t2.mps	**3600.09**	3600.11	**92141.89**	116211.18	0.00	0.00	0.80	**1.20**
fiball.mps	3084.24	**1705.33**	**16081.13**	16625.20	0.20	**1.00**	**0.60**	0.40
gen-ip002.mps	**3600.08**	3600.10	**490285.45**	994279.08	0.00	0.00	0.20	**2.00**
gen-ip054.mps	3600.09	**3600.08**	**457148.67**	993144.34	0.00	0.00	0.20	**5.20**
germanrr.mps	3600.38	**3600.33**	2994.52	**2976.11**	0.00	0.00	0.20	**0.40**
glass-sc.mps	3600.26	**3600.24**	**70977.51**	78388.83	0.00	0.00	0.00	0.00
glass4.mps	3600.09	**3367.85**	**630519.75**	904667.44	0.00	**0.20**	0.40	**22.80**
gmu-35-40.mps	3600.10	**3600.08**	**577557.38**	852713.65	0.00	0.00	0.40	**16.40**
gmu-35-50.mps	**3600.13**	3600.15	**325272.13**	582101.45	0.00	0.00	0.40	**13.80**
graph20-20-1rand.mps	3600.21	**3600.11**	**35823.18**	38320.32	0.00	0.00	**0.60**	0.00
graphdraw-domain.mps	3600.08	3600.08	**340491.40**	843202.13	0.00	0.00	1.20	**10.80**
h80x6320d.mps	192.03	**165.01**	7.00	7.00	1.00	1.00	0.00	0.00
ic97_potential.mps	3600.10	**3600.09**	**251713.56**	673577.59	0.00	0.00	0.80	**12.60**
icir97_tension.mps	3600.16	**3600.13**	**190907.50**	355227.55	0.00	0.00	0.60	**6.60**
irp.mps	**19.40**	20.13	3.00	3.00	1.00	1.00	0.00	0.00
istanbul-no-cutoff.mps	342.72	**323.77**	456.15	456.15	1.00	1.00	0.00	0.00
lectsched-5-obj.mps	3600.54	**3600.44**	**37539.23**	60741.38	0.00	0.00	2.60	**4.00**
leo1.mps	3600.38	**3600.23**	**40954.42**	97533.33	0.00	0.00	**2.40**	0.40
leo2.mps	**3600.36**	3600.46	**17470.85**	34214.10	0.00	0.00	**1.40**	0.00
lotsize.mps	3600.10	**3600.08**	**4352.55**	4902.74	0.00	0.00	0.60	**4.20**
mad.mps	3600.08	**3600.05**	**1115036.96**	1294607.08	0.00	0.00	1.00	**11.80**
markshare2.mps	**3600.07**	3600.11	**4873115.59**	5179318.15	0.00	0.00	0.40	0.40
markshare_4_0.mps	**3509.23**	3600.07	**1351789.23**	1758934.83	**0.20**	0.00	0.40	0.40
mas74.mps	3600.08	**3600.07**	**406537.79**	1045392.00	0.00	0.00	0.00	**2.60**
mas76.mps	3128.70	**1645.80**	**414093.05**	537679.62	0.50	**1.00**	0.25	**0.75**
mc11.mps	**3600.08**	3600.18	**81497.76**	114280.48	0.00	0.00	0.00	**5.80**
mcsched.mps	**239.83**	241.30	19553.08	19553.08	1.00	1.00	0.00	0.00
mik-250-20-75-4.mps	1071.36	**894.57**	**173467.27**	193867.84	1.00	1.00	0.00	**0.50**
milo-v12-6-r2-40-1.mps	3600.13	**3600.11**	**30083.46**	48969.03	0.00	0.00	0.50	**8.75**
momentum1.mps	3600.35	**3600.24**	**2794.34**	2972.13	0.00	0.00	1.25	**6.50**
mushroom-best.mps	3600.43	**3600.37**	**6233.12**	19358.33	0.00	0.00	**0.40**	0.20
mzzv11.mps	294.80	**279.10**	313.46	**312.08**	1.00	1.00	0.20	0.20
mzzv42z.mps	**161.65**	162.32	14.53	14.53	1.00	1.00	0.00	0.00
n2seq36q.mps	3600.47	**3600.44**	**49254.92**	73065.07	0.00	0.00	1.40	**1.60**
n3div36.mps	**3600.77**	3600.80	**18511.51**	22336.79	0.00	0.00	0.00	**0.60**
n5-3.mps	47.84	**46.18**	1418.45	1418.45	1.00	1.00	0.00	0.00
neos-1171448.mps	57.15	**52.17**	94.05	94.05	1.00	1.00	0.00	0.00
neos-1171737.mps	3600.23	**3600.12**	**41179.00**	73291.76	0.00	0.00	0.00	**0.80**
neos-1445765.mps	79.29	**75.25**	4.53	4.53	1.00	1.00	0.00	0.00
neos-1456979.mps	**3600.16**	3600.19	**96850.50**	112603.56	0.00	0.00	0.20	**2.80**
neos-1582420.mps	26.21	**24.74**	65.52	65.52	1.00	1.00	0.20	0.20
neos-2657525-crna.mps	3600.16	**3600.09**	**556806.62**	750373.25	0.00	0.00	0.75	**2.25**
neos-2746589-doon.mps	**3600.74**	3601.05	9551.22	**8525.94**	0.00	0.00	0.80	**2.40**
neos-3004026-krka.mps	**317.48**	328.86	20465.75	20465.75	1.00	1.00	0.00	0.00
neos-3024952-loue.mps	3600.15	**3600.06**	**165185.21**	208663.38	0.00	0.00	0.00	**22.67**
neos-3046615-murg.mps	**3600.06**	3600.08	**121905.30**	820139.32	0.00	0.00	2.60	**16.40**
neos-3083819-nubu.mps	63.37	**50.59**	6344.27	**5909.95**	1.00	1.00	1.20	1.20

Continued on next page ...

Table B.2 – continued from previous page

Instance	Total Time		Total Nodes		Fraction Solved		Success Count	
	w/o SLNS	w SLNS	w/o SLNS	w SLNS	w/o SLNS	w SLNS	RINS	SLNS
neos-3216931-puriri.mps	**3600.14**	3600.16	**2337.57**	2419.14	0.00	0.00	0.00	0.00
neos-3381206-awhea.mps	**6.20**	6.43	142.00	142.00	1.00	1.00	0.00	0.00
neos-3627168-kasai.mps	**3600.11**	3600.17	**242116.16**	585096.73	0.00	0.00	2.80	**14.80**
neos-3656078-kumeu.mps	**3600.16**	3600.21	8710.00	**5239.00**	0.00	0.00	2.00	**5.00**
neos-4300652-rahue.mps	**3600.81**	3600.91	**994.63**	1010.06	0.00	0.00	0.00	0.00
neos-4338804-snowy.mps	3600.11	3600.11	**173090.90**	529973.23	0.00	0.00	3.40	**9.80**
neos-4387871-tavua.mps	**3600.09**	3600.15	**30279.21**	38569.48	0.00	0.00	0.60	**8.00**
neos-4722843-widden.mps	**3600.75**	3600.82	2026.00	**2020.00**	0.00	0.00	0.00	**2.00**
neos-4738912-atrato.mps	1070.18	**519.59**	40484.60	**24748.27**	1.00	1.00	1.00	**6.20**
neos-4763324-toguru.mps	3601.17	**3600.87**	**3837.69**	4712.51	0.00	0.00	0.40	0.40
neos-4954672-berkel.mps	3600.18	**3600.09**	**329482.40**	495201.87	0.00	0.00	0.00	**9.40**
neos-5107597-kakapo.mps	**3600.15**	3600.17	**126913.45**	183648.23	0.00	0.00	0.00	**17.60**
neos-5188808-nattai.mps	1966.34	**1902.64**	19564.54	**19318.90**	1.00	1.00	0.00	**1.00**
neos-5195221-niemur.mps	3600.38	**3413.26**	19466.73	**19235.71**	0.00	**0.20**	1.80	**7.20**
neos-662469.mps	**3600.36**	3600.53	**9227.41**	21665.03	0.00	0.00	3.20	**4.00**
neos-873061.mps	**3601.73**	3605.29	1406.73	**1389.73**	0.00	0.00	**2.60**	0.00
neos-911970.mps	**3600.09**	3600.25	**428650.26**	808101.88	0.00	0.00	0.60	**6.60**
neos-933966.mps	**207.41**	209.39	26.49	26.49	1.00	1.00	0.00	0.00
neos-950242.mps	**3600.24**	3600.25	**932.14**	1299.04	0.00	0.00	0.00	0.00
neos-957323.mps	58.66	**58.50**	3.00	3.00	1.00	1.00	0.00	0.00
neos-960392.mps	297.78	**259.13**	129.38	129.38	1.00	1.00	0.00	0.00
neos17.mps	95.69	**41.12**	8142.79	**7337.97**	1.00	1.00	0.80	**2.60**
neos5.mps	3600.12	**3600.06**	**397930.18**	946703.76	0.00	0.00	**1.00**	0.00
net12.mps	941.91	**866.84**	**2053.48**	2081.68	1.00	1.00	0.40	0.40
nexp-150-20-8-5.mps	3600.20	**3600.15**	**8309.66**	13826.66	0.00	0.00	**1.40**	0.00
ns1208400.mps	580.30	**469.86**	5355.72	5355.72	1.00	1.00	0.00	0.00
ns1830653.mps	1344.91	**733.32**	**22825.68**	26280.82	0.80	**1.00**	0.80	**1.60**
ns1952667.mps	1657.27	**1302.45**	4287.21	4287.21	1.00	1.00	0.00	0.00
nu25-pr12.mps	7.49	**6.38**	18.61	18.61	1.00	1.00	0.00	0.00
nursesched-sprint02.mps	65.07	**55.52**	78.67	78.67	1.00	1.00	0.00	0.00
opm2-z10-s4.mps	**3600.73**	3600.98	141.49	**96.11**	0.00	0.00	0.00	0.00
p200x1188c.mps	**6.90**	7.22	3.00	3.00	1.00	1.00	0.00	0.00
peg-solitaire-a3.mps	**2483.56**	2830.09	2009.00	2009.00	1.00	1.00	0.00	0.00
pg.mps	21.24	**20.87**	545.62	**539.69**	1.00	1.00	0.40	0.40
pg5_34.mps	2965.45	**1670.95**	**129940.19**	134292.25	0.40	**1.00**	0.80	**4.60**
physicliansched6-2.mps	**119.69**	121.44	81.36	81.36	1.00	1.00	0.00	0.00
piperout-08.mps	270.05	**264.31**	918.91	**862.24**	1.00	1.00	0.00	**0.80**
piperout-27.mps	361.10	**353.10**	427.90	427.90	1.00	1.00	**0.20**	0.00
pk1.mps	1546.83	**1080.30**	**334642.60**	337004.62	1.00	1.00	1.00	**5.00**
radiationm18-12-05.mps	**3600.34**	3600.39	**118158.72**	159340.03	0.00	0.00	2.40	**15.40**
radiationm40-10-02.mps	3601.52	**3601.32**	**8241.84**	11768.41	0.00	0.00	0.20	**7.80**
rail01.mps	3600.80	3600.80	20.00	20.00	0.00	0.00	0.00	0.00
rail507.mps	138.34	**121.40**	776.22	776.22	1.00	1.00	0.00	0.00
ran14x18-disj-8.mps	3600.11	**3600.05**	**343795.94**	484604.14	0.00	0.00	0.80	**22.40**
reblock115.mps	**3600.12**	3600.13	**150356.25**	227933.45	0.00	0.00	3.80	**17.80**
rmatr100-p10.mps	100.55	**89.62**	904.60	904.60	1.00	1.00	**0.20**	0.00
rmatr200-p5.mps	3600.40	**3600.34**	**861.89**	1095.37	0.00	0.00	0.00	**0.25**
rocI-4-11.mps	164.10	**156.81**	17190.87	17190.87	1.00	1.00	**0.40**	0.00
rocII-5-11.mps	3600.52	3600.52	**26775.93**	34168.43	0.00	0.00	0.00	**2.20**
rococoB10-011000.mps	3600.13	**3600.10**	**116776.42**	119125.92	0.00	0.00	1.60	**8.00**
rococoC10-001000.mps	3600.13	**3600.10**	**226200.38**	273425.58	0.00	0.00	0.60	**15.80**
roll3000.mps	2568.34	**2231.83**	**139905.20**	149362.74	1.00	1.00	1.80	**8.00**
satellites2-40.mps	**3600.42**	3600.60	125.00	**66.00**	0.00	0.00	0.00	0.00
satellites2-60-fs.mps	3600.42	**3600.22**	**163.00**	183.00	0.00	0.00	0.00	0.00
sct2.mps	3600.11	**3600.09**	**77337.68**	263616.98	0.00	0.00	1.00	**3.00**
seymour.mps	**3600.12**	3600.16	**41626.45**	52490.32	0.00	0.00	1.40	**4.20**
seymour1.mps	284.40	**277.75**	**2232.08**	2241.02	1.00	1.00	0.20	**1.00**
sing326.mps	**3600.63**	3600.82	**5217.53**	6068.87	0.00	0.00	2.00	**6.60**
sing44.mps	**3600.67**	3601.02	**5625.18**	5803.36	0.00	0.00	1.20	**4.80**
snp-02-004-104.mps	**3602.57**	3602.59	154.47	**121.60**	0.00	0.00	0.00	0.00
sp150x300d.mps	**3600.08**	3600.14	**168410.00**	697981.00	0.00	0.00	0.00	0.00
sp97ar.mps	**3600.65**	3600.74	**10606.34**	31965.33	0.00	0.00	**3.00**	0.20
sp98ar.mps	3600.80	**3600.53**	**15187.13**	31137.32	0.00	0.00	**3.80**	0.00
supportcase18.mps	**3600.21**	3600.26	**37333.82**	97209.07	0.00	0.00	**0.60**	0.00
supportcase26.mps	3600.14	**3600.12**	**223687.86**	695869.30	0.00	0.00	0.40	**10.40**
supportcase33.mps	2033.37	**1784.52**	8045.60	**7678.71**	**1.00**	0.80	0.00	**0.60**
supportcase40.mps	3600.38	**3501.44**	**12947.05**	37884.52	0.00	**0.60**	0.40	**6.60**
supportcase42.mps	3601.17	**3600.77**	**7525.02**	12619.45	0.00	0.00	0.50	**2.00**
swath1.mps	75.84	**70.60**	880.81	**876.38**	1.00	1.00	0.25	0.25
swath3.mps	3515.57	**2967.56**	**39791.72**	81306.08	0.20	0.20	0.80	**1.40**
tbfp-network.mps	1409.77	**1360.22**	100.06	100.06	1.00	1.00	0.00	0.00
thor50dday.mps	**3601.11**	3602.31	**1022.14**	1584.67	0.00	0.00	**0.50**	0.00
timtab1.mps	3600.09	**3521.75**	**452158.92**	662145.91	0.00	**0.20**	1.60	**12.60**
tr12-30.mps	**3600.05**	3600.09	**336947.43**	375019.26	0.00	0.00	0.60	**7.80**
traininstance2.mps	**3600.16**	3600.24	**6714.12**	7170.22	0.00	0.00	0.60	**0.80**
traininstance6.mps	3600.14	3600.14	28588.93	**26454.05**	0.00	0.00	0.20	**2.00**
trento1.mps	3600.31	**3600.20**	**16268.06**	22435.67	0.00	0.00	0.60	**9.00**
uccase12.mps	**3601.32**	3601.42	**5077.85**	9655.18	0.00	0.00	0.60	**1.00**
uccase9.mps	3600.77	**3600.61**	**1039.08**	1220.45	0.00	0.00	0.00	**1.20**
uct-subprob.mps	3600.19	**3600.09**	**170054.68**	189334.80	0.00	0.00	1.00	**7.20**
unitcal_7.mps	232.51	**227.99**	240.30	240.30	1.00	1.00	0.00	0.00
wachplan.mps	**3600.21**	3600.26	**39473.64**	92073.21	0.00	0.00	0.00	0.00