Algorithms and Machine Learning for fair and classical combinatorial optimization

Thi Quynh Trang Vo

THÈSE POUR OBTENIR LE DIPLÔME DE DOCTEUR DE L'UNIVERSITÉ CLERMONT AUVERGNE

En Informatique

École Doctorale Sciences Pour l'Ingénieur (ED SPI)

Algorithms and Machine Learning for fair and classical combinatorial optimization

Présenté par **Thi Quynh Trang VO**

Date de la soutenance - Le 13 mai 2024

Sous la supervision des

Viet Hung NGUYEN et Mourad BAIOU

Devant le jury composé de

Dritan NACE	Professeur à l'Université de Technologie de Compiègne	Rapporteur
Axel PARMENTIER	Chercheur HDR à l'Ecole Nationale des Ponts et Chaussées	Rapporteur
Sophie DEMASSEY	MCF HDR à Ecole de Mines de Paris	Examinatrice
Kim Thang NGUYEN	Professeur à l'Université Grenoble Alpes	Examinateur
Fatiha BENDALI	Professeure à l'Université Clermont Auvergne	Examinatrice
Mourad BAIOU	Directeur de recherche au CNRS, LIMOS UMR 6158	Co-encadrant
Viet Hung NGUYEN	Professeur à l'Université Clermont Auvergne	Co-encadrant, directeur de thèse

UNIVERSITÉ Clermont Auvergne

LIMOS

Acknowledgements

First, I would like to express my sincere gratitude to Professor Viet Hung Nguyen for the invaluable mentorship, guidance, and inspiration he provided throughout my time as his PhD student. I sincerely appreciate all I have learned from him, both in research and life lessons. His endless support and encouragement made this work possible.

I am also very grateful to Professor Mourad Baiou for his priceless mentorship and support. He always gave me valuable guidance at critical junctures, especially at the end of my PhD journey. His recognition of my efforts made me feel motivated to keep going.

Another person who had a significant impact on my work is Professor Paul Weng. Through many illuminating discussions with him, I gained invaluable knowledge and feel extremely fortunate to have had the opportunity to collaborate with him. I would also like to express my gratitude to Professor Ngoc Chi Le. Working with him over the last four years has completely transformed my perspective on research.

Coming from a distant country with significant differences in language and culture, there were many things I had to learn from the beginning. Fortunately, I received tremendous help from my colleagues in the lab. First, I want to express my gratitude to Aurelien Mombelli. He started his PhD the same day as me and has been multi-functional, serving as my French teacher, administrative "secretary," and drinking buddy. I also want to thank my patient colleagues who spoke French with me and taught me so much: Alexey, Caroline, Elodie, Simon, Malex, Amal, Mari, Chijia, Jose, Romain, Pedro, Sofian, Tam, and others. I am grateful to the LIMOS laboratory for hosting me and providing a supportive working environment. I also thank the staff who assisted me greatly: Beatrice, Martine, Bastien, and Valerie. Finally, I appreciate the advice and encouragement I received from LIMOS professors: Vincent, Fatiha, Christian, Herve, and more.

I am deeply grateful for the community of Vietnamese friends who enriched my experience as a PhD student in France. First, I want to express my heartfelt appreciation to my elder brothers, Minh and Trung. Your care and support gave me strength during those early transition days when I first arrived in France. Additionally, I owe tremendous thanks to my dearest friends - Hieu, Loan, Thao, Phuong Anh, Tri Minh, Hoang, and Ngoc Minh. You stood by my side, helping me and sharing my joys and sorrows. I could not have finished this thesis without you. Finally, I am also grateful to Tung for his invaluable help in tutoring me on scientific writing and for the significant mental support he offered. His guidance bolstered me tremendously along this journey.

Last but not least, I am deeply thankful for my family, whose unconditional love and steadfast support have nurtured me. Their unwavering belief in me gives me strength and provides a comforting home to come back to, no matter where my path leads.

Résumé

L'optimisation combinatoire est un domaine des mathématiques dans lequel un problème consiste à trouver une solution optimale dans un ensemble fini d'objets. Elle a des applications cruciales dans de nombreux domaines, notamment les mathématiques appliquées, le génie logiciel, l'informatique théorique et l'apprentissage automatique. Le branch-and-cut est l'un des algorithmes les plus utilisés pour résoudre exactement des problèmes d'optimisation combinatoire. Dans cette thèse, nous nous concentrons sur les aspects informatiques du branch-and-cut et plus particulièrement, sur deux aspects importants de l'optimisation combinatoire : *l'équité des solutions* et *l'intégration de l'apprentissage automatique*.

Dans la partie I (chapitres 3 et 4), nous étudions deux approches courantes pour traiter la question de l'équité dans l'optimisation combinatoire, qui a fait l'objet d'une attention particulière au cours des dernières décennies. La première approche est *l'optimisation combinatoire équilibrée*, qui trouve une solution équitable en minimisant la différence entre les plus grands et les plus petits composants utilisés. En raison des difficultés à délimiter ces composants, à notre connaissance, aucun cadre général exact basé sur la programmation linéaire en nombres entiers mixtes (MILP) n'a été proposé pour l'optimisation combinatoire équilibrée. Pour combler cette lacune, nous présentons au chapitre 3 une nouvelle classe de plans de coupe locaux adaptés aux problèmes d'optimisation combinatoire équilibrée pour l'algorithme du branch-and-cut. Nous démontrons l'efficacité de la méthode proposée dans la carde du problème du voyageur de commerce équilibré. Notamment, nous introduisons des algorithmes pour la recherche de bornes et des mécanismes pour la détermination des variables afin d'accélérer un peu plus les performances.

Une deuxième approche pour traiter la question de l'équité est l'optimisation combinatoire *Ordered Weighted Average (OWA)*, qui consiste à utiliser l'opérateur OWA dans la fonction objectif. En raison de l'opérateur d'ordonnancement, l'optimisation combinatoire OWA est non linéaire, même si ses contraintes d'origine sont linéaires. Deux formulations MILP de tailles différentes ont été introduites dans la littérature pour linéariser l'opérateur OWA. Cependant, la formulation la plus performante pour l'optimisation combinatoire OWA reste incertaine, car l'intégration des méthodes de linéarisation peut introduire des difficultés supplémentaires. Dans le chapitre 4, nous fournissons des comparaisons théoriques et empiriques des deux formulations MILP pour l'optimisation combinatoire OWA. En particulier, nous prouvons que les formulations sont équivalentes en termes de relaxation de programmation linéaire. Nous montrons empiriquement que pour les problèmes d'optimisation combinatoire OWA, la for-

mulation avec le plus de variables peut être résolue plus rapidement avec le branch-and-cut.

Dans la partie II (chapitre 5), nous développons des méthodes d'application de l'apprentissage automatique pour améliorer les problèmes de décision fondamentaux du branch-and-cut, en mettant l'accent sur la génération de coupes. Ce dernier problème se réfère à la décision de générer des coupes ou des branches à chaque nœud de l'arbre de recherche. Nous démontrons empiriquement que cette décision a un impact significatif sur les performances du branch-and-cut, en particulier pour les coupes combinatoires qui exploitent les faces de la coque convexe des solutions réalisables. Nous proposons ensuite un cadre général combinant l'apprentissage supervisé et l'apprentissage par renforcement afin d'apprendre des stratégies efficaces pour générer des coupes combinatoires dans la méthode branch-and-cut. Notre cadre comporte deux composantes : un détecteur de coupes pour prédire l'existence de coupes et un évaluateur de coupes pour choisir entre la génération de coupes et le branchement. Enfin, nous fournissons des résultats expérimentaux montrant que la méthode proposée est plus performante que les stratégies couramment utilisées pour la génération de coupes, même sur des instances plus grandes que celles utilisées pour l'apprentissage.

Abstract

Combinatorial optimization is a field of mathematics that searches for an optimal solution in a finite set of objects. It has crucial applications in many fields, including applied mathematics, software engineering, theoretical computer science, and machine learning. *Branch-and-cut* is one of the most widely-used algorithms for solving combinatorial optimization problems exactly. In this thesis, we focus on the computational aspects of branch-and-cut when studying two critical dimensions of combinatorial optimization: *the fairness of solutions* and *the integration of machine learning*.

In Part I (Chapters 3 and 4), we study two common approaches to deal with the issue of fairness in combinatorial optimization, which has gained significant attention in the past decades. The first approach is *balanced combinatorial optimization*, which finds a fair solution by minimizing the difference between the largest and smallest components used. Due to the difficulties in bounding these components, to the best of our knowledge, no general exact framework based on mixed-integer linear programming (MILP) has been proposed for balanced combinatorial optimization. To address this gap, in Chapter 3, we present a branch-and-cut algorithm and a novel class of local cutting planes tailored for balanced combinatorial optimization problems. We demonstrate the effectiveness of the proposed framework in the Balanced Traveling Salesman Problem. Additionally, we introduce bounding algorithms and mechanisms to fix variables to accelerate performance further.

The second approach to handling the issue of fairness is *Ordered Weighted Average (OWA) combinatorial optimization*, which integrates the OWA operator into the objective function. Due to the ordering operator, OWA combinatorial optimization is nonlinear, even if its original constraints are linear. Two MILP formulations of different sizes have been introduced in the literature to linearize the OWA operator. However, which formulation performs best for OWA combinatorial optimization remains uncertain, as integrating the linearization methods may introduce additional difficulties. In Chapter 4, we provide theoretical and empirical comparisons of the two MILP formulations for OWA combinatorial optimization. In particular, we prove that the formulations are equivalent in terms of the linear programming relaxation. We empirically show that for OWA combinatorial optimization problems, the formulation with more variables can be solved faster with branch-and-cut.

In Part II (Chapter 5), we develop methods for applying machine learning to enhance fundamental decision problems in branch-and-cut, with a focus on cut generation. Cut generation refers to the decision of whether to generate cuts or to branch at each node of the search tree. We

empirically demonstrate that this decision significantly impacts branch-and-cut performance, especially for combinatorial cuts that exploit the facial structure of the convex hull of feasible solutions. We then propose a general framework combining supervised and reinforcement learning to learn effective strategies for generating combinatorial cuts in branch-and-cut. Our framework has two components: a cut detector to predict cut existence and a cut evaluator to choose between generating cuts and branching. Finally, we provide experimental results showing that the proposed method outperforms commonly used strategies for cut generation, even on instances larger than those used for training.

Keywords: Combinatorial optimization, Mixed-Integer Linear Programming, Branch-and-Cut, Balanced combinatorial optimization, Ordered Weighted Average, Machine Learning, Cut generation, Combinatorial cuts.

List of publications

1. **Thi Quynh Trang Vo**, Mourad Baiou, Viet Hung Nguyen and Paul Weng. *Learning to Cut Generation in Branch-and-Cut algorithms for Combinatorial Optimization*. In: submitted to the ACM Transactions on Evolutionary Learning and Optimization, 2024.

2. **Thi Quynh Trang Vo**, Mourad Baiou and Viet Hung Nguyen. *A Branch-and-Cut algorithm for the Balanced Traveling Salesman Problem.* In: Journal of Combinatorial Optimization 47, 4, 2024. DOI: https://doi.org/10.1007/s10878-023-01097-4

3. **Thi Quynh Trang Vo**, Mourad Baiou, Viet Hung Nguyen and Paul Weng. *Improving Subtour Elimination Constraint Generation in Branch-and-Cut Algorithms for the TSP with Machine Learning.* In: Proceedings of the 17th Learning and Intelligent Optimization Conference (LION). Lecture Notes in Computer Science (LNCS). Springer, 2023. DOI: https://doi.org/10.1007/978-3-031-44505-7_36.

4. **Thi Quynh Trang Vo**, Mourad Baiou, Viet Hung Nguyen and Paul Weng. *A comparative study of linearization methods for Ordered Weighted Average.* In: Proceedings of the 12th International Workshop on Resilient Networks Design and Modeling (RNDM), 2022. DOI: https://doi.org/10.1109/RNDM55901.2022.9927720

5. Minh Hieu Nguyen, Mourad Baiou, Viet Hung Nguyen and **Thi Quynh Trang Vo**. *Generalized Nash Fairness solutions for Bi-Objective Minimization Problems.* In: Networks Journal, 2023. DOI: https://doi.org/10.1002/net.22182

6. Minh Hieu Nguyen, Mourad Baiou, Viet Hung Nguyen, **Thi Quynh Trang Vo.** *Proportional Fairness for Combinatorial Optimization.* In: Latin American Theoretical Informatics Symposium (LATIN 2024), Mar 2024, Puerto Varas, Chile.

7. Minh Hieu Nguyen, Mourad Baiou, Viet Hung Nguyen and **Thi Quynh Trang Vo.** *Nash fairness solutions for balanced TSP.* In: Proceedings of the 10th International Network Optimization Conference (INOC), 2022. DOI: http://dx.doi.org/10.48786/inoc.2022.17

Contents

Chapter 1

General introduction

Combinatorial optimization is a subfield of mathematical optimization that involves finding an optimal solution from a finite set of possibilities. It is one of the most active research areas in recent years since thousands of real-life decision problems that significantly impact our daily lives can be formulated as combinatorial optimization problems. Examples include optimizing resource plans in healthcare [3], minimizing the global transportation costs of a delivery schedule [4], minimizing the emissions of greenhouse gases of logistics operations [5], and maximizing the total profit of project portfolio selection in business [6]. These problems are often characterized by their complexity and difficulty, with many classified as NP-hard, remaining challenges to achieve optimal or even near-optimal solutions.

While the possible solutions of combinatorial optimization are finite, their number increases exponentially with the instance size, rendering exhaustive search intractable. To overcome this challenge, researchers have developed efficient search algorithms tailored to specific combinatorial optimization problems. However, these algorithms typically demand an in-depth understanding of the problem's characteristics and a substantial investment in time and resources. An alternative approach is establishing generic methods to solve various combinatorial optimization problems.

One such generic method is mixed-integer linear programming (MILP), a powerful technique for modeling problems. Most combinatorial optimization problems can be formulated naturally as MILP formulations where decisions are represented by variables that can take either continuous or discrete values. The relationships between these variables are defined through linear constraints, and the goal is to optimize a linear objective function subject to these constraints.

Once a combinatorial optimization problem is formulated as an MILP problem, it can be solved using MILP solvers. The backbone of the state-of-the-art modern MILP solvers is *branch-and-cut*, which merges two well-known methods: branch-and-bound and cutting plane. While branch-and-bound recursively partitions the search space into smaller subspaces and bounds the objective function in each subspace, the cutting plane method iteratively adds valid inequalities to refine the search space. Combining these two methods allows branch-and-cut to prune the solution space effectively and quickly converge to the optimal solution.

While powerful, the empirical performance of branch-and-cut is irregular and sensitive,

enormously depending on problem attributes and implementation details. For example, a minor change in the objective function or the way of representing constraints can lead to a dramatic fluctuation in performance. Factors of the implementation, like the internal selection strategies, the use of heuristics, or the choices of linear programming (LP) algorithms, can also impact the branch-and-cut effectiveness.

Motivated by these issues, this thesis aims to bridge the gap in knowledge surrounding the computational aspects of branch-and-cut when solving combinatorial optimization problems.

1.1 Research questions and scope

In this thesis, we focus on studying branch-and-cut in the context of two critical dimensions of combinatorial optimization: *the fairness of solutions* and *the integration of machine learning*.

1.1.1 Fair combinatorial optimization

Fairness (*equity*) is a fundamental concept that has intrigued humans for centuries. It is something that we all desire and strive for, as it forms the foundation of our social interactions and relationships. The idea of fairness is not limited to any specific context or situation; rather, it is a universal concept that appears in all areas of life. For example, during times of crisis, it is essential to ensure the equitable distribution of sparse resources, such as food and medicine, so that every individual has access to the necessities of life. In another context, such as in a classroom setting, all students should be treated uniformly and without discrimination based on factors such as their wealth, abilities, or appearance.

Due to its nature, the issue of fairness has also received considerable attention from researchers in combinatorial optimization. This issue arises naturally in combinatorial optimization problems. For example, in the assignment problem that assigns tasks to workers, a solution in which some workers receive significantly larger workloads than others could be perceived as unfair treatment.

Various approaches have been proposed in the literature to address the issue of fairness in combinatorial optimization with different ways to model fairness or equity. In this thesis, we focus on two popular approaches: *Balanced combinatorial optimization* [7] and *Ordered Weighted Average (OWA) combinatorial optimization* [8].

1.1.1.1 Balanced combinatorial optimization

Balanced combinatorial optimization, proposed by Martello et al. [7], finds a fair solution by minimizing the difference in values between the most expensive and least expensive components; we call this difference *the max-min distance*, for abbreviation. This approach was first defined in the context of the assignment problem [7] and then extended and generalized for other special cases of combinatorial optimization due to its intuitive nature [7, 9–19].

There are two main approaches for solving balanced combinatorial optimization. The first is directly constructing a solution with the smallest max-min distance by exploiting problem-specific structures. This approach focused primarily on polynomial-solvable problems. The second is based on the double-threshold algorithm, a general iterative framework finding

thresholds of the largest and smallest components. These thresholds can be found by repeatedly verifying the existence of a feasible solution in a given set through the so-called *feasibility subroutine*. Consequently, this algorithm's complexity depends on the feasibility subroutine, which is sometimes NP-hard. Variants aim to reduce the complexity of feasibility verification problems and the number of iterations needed.

To the best of our knowledge, no generic algorithm based on MILP has been proposed for balanced combinatorial optimization despite MILP successfully tackling a wide range of problems. The reason is that the primary challenge when solving balanced combinatorial optimization is not in formulating problems but in bounding the largest and smallest components. When these components are not tightly bound, it becomes much more difficult to prune nodes in the enumeration tree during the branch-and-cut process. As a result, solving balanced combinatorial optimization problems by branch-and-cut can be incredibly time-consuming and inefficient.

Research question 1: *How do we design branch-and-cut to solve balanced combinatorial optimization efficiently?*

To answer this question, in Chapter 3, we propose a special-purpose branch-and-cut algorithm for balanced combinatorial optimization. The central of our algorithm is a new class of local cutting planes, called *local bounding cuts*. These cutting planes require no problem-specific structure and instead leverage branch-and-cut tree information to better bound the smallest component cost. We apply the proposed branch-and-cut to solve the balanced traveling salesman problem (BTSP), an NP-hard case which was only solved heuristically [13] in the literature. To further accelerate performance, we develop algorithms to initialize lower and upper bounds on the optimal value of the BTSP and additional mechanisms to tighten the largest and smallest edge costs. Computational results on the same TSPLIB testbed as in [13] show that our proposed approach can solve 63 out of 65 instances to proven optimality, while the double-threshold based heuristics [13] only certify the optimality of 27 solutions.

1.1.1.2 OWA combinatorial optimization

Notice that balanced combinatorial optimization focuses only on the max-min distance, which may lead to inefficient solutions regarding the total outcome. A more sophisticated approach is *OWA combinatorial optimization* [8], whose objective function integrates the OWA operator [20]. The intuition behind OWA combinatorial optimization derives from an observation that when dealing with solution fairness, we care about the set of component values without considering which component takes a specific value. Thus, this approach considers each component as an individual objective and aggregates objectives by the OWA operator. OWA combinatorial optimization solutions are efficient in the sense of Pareto-optimal, accounting for inequality minimization according to the Pigou-Dalton approach. This OWA-based method is well-suited for real-life situations requiring appropriate trade-offs between fairness and efficiency.

Due to the ordering operator in the objective function, OWA combinatorial optimization problems are non-linear, even if their original constraints are linear. Fortunately, there exist two linearization methods for the OWA with decreasing weights. The first was proposed by Orgyczak et al. [1], which reformulates the OWA as a combination of Lorenz components and

represents each Lorenz component as a dual of a linear program. The second was introduced by Chassein et al. [2], which uses the permutahedron as the OWA value is the maximum among all permutations of the inner product between the OWA weights and solution components. When comparing in terms of size, Chassein's method uses fewer variables than Orgyczak's. Moreover, when applied to several continuous optimization problems, Chassein's formulation can be solved more quickly [2]. However, in OWA combinatorial optimization, integrating linearization methods into the problem formulation can cause additional difficulties. Therefore, questions remain about the comparison and relationship between the two methods in OWA combinatorial optimization.

Research question 2: *Is there any relation between the two linearization methods for OWA combinatorial optimization? Is the formulation with fewer variables always solved faster by branch-and-cut?*

To address these questions, in Chapter 4, we conduct a comprehensive comparative study between the two linearization methods for OWA in the context of combinatorial optimization in both theoretical and empirical aspects. We prove that, despite employing different numbers of variables, the two formulations are equivalent in terms of linear relaxations. Interestingly, Orgyczak's formulation, which uses more variables, can be solved faster than Chassein's when applied to the OWA traveling salesman problem.

1.1.2 Machine Learning for Combinatorial Optimization

Branch-and-cut is notoriously highly configurable. Its performance heavily depends on the configuration of the inner decision strategies, such as variable selection, node selection, cut selection, or cut generation. A deft configuration can help branch-and-cut solve computationally challenging problems. However, identifying an effective configuration is a difficult challenge.

In practical situations, we usually repeatedly solve a combinatorial optimization problem with many different but related instances. This process generates a vast of historical data that can contain meaningful patterns. A natural idea is to leverage these patterns to solve new instances faster. However, manually extracting these patterns requires significant expert effort. Therefore, an automated tool is needed to systematically discover and leverage patterns for constructing deft decision policies.

Recently, machine learning has emerged as a promising tool for automatically learning efficient branch-and-cut strategies from past solving experience. Machine learning algorithms can identify patterns in data and improve performance across various domains. Compared to manual-tuning heuristics, machine learning has two key advantages. Firstly, this approach is more systematic, potentially resulting in more effective heuristics. Secondly, machine learning enables rapidly developing specialized strategies even for new, unfamiliar problem classes, with less reliance on human domain knowledge.

Taking these advantages, researchers have applied machine learning to learn policies for branch-and-cut decision problems, including variable selection [21, 22], node selection [23], and cut selection [24], and obtained promising results. However, there is still an important issue that has not been fully addressed - *cut generation*, a key design challenge when combining branch-and-bound and cutting plane methods. Cut generation refers to the problem of de-

ciding whether to generate cuts or to branch at nodes of the branch-and-bound tree to reduce overall runtime. This problem dramatically impacts branch-and-cut performance. Indeed, generating cuts can significantly reduce the tree size but can also damage the overall runtime due to solving separation problems and LP relaxations.

Furthermore, previous machine learning works on cut-related decision problems have focused primarily on general-purpose cuts, which are based on the integrality conditions of variables. Combinatorial cuts, a crucial class of cuts encoding problem-specific structures, have yet to receive much attention. The lack of studies on generating combinatorial cuts is a notable research gap, as combinatorial cuts are indispensable when solving combinatorial optimization, especially NP-hard problems.

Research question 3: *How can we use machine learning for learning a generation policy for combinatorial cuts?*

To address this question, in Chapter 5, we propose a general machine-learning-based framework to learn generation strategies for combinatorial cuts. Our framework is versatile and applicable to various combinatorial optimization problems with different types of cuts. Moreover, the trained policies are adaptable to arbitrary-sized instances, even though they are initially trained with fixed-size examples. Experiments on two well-known classes of combinatorial cuts, i.e., subtour elimination constraints for the traveling salesman problem and cycle inequalities for the max-cut problem, show that our approach can significantly accelerate the branch-and-cut performance.

Chapter 2

Background

Summary

2.1 Combinatorial optimization

Combinatorial optimization involves maximizing or minimizing an objective function over variables subject to constraints and integrality restrictions. In this thesis, we focus on a critical class of combinatorial optimization where the objective and constraints are linear, and all variables are binary (restricted to 0 or 1). More precisely, we consider combinatorial optimization problems of the form:

$$\text{(IP)} \quad \min \quad \boldsymbol{c}^T \boldsymbol{x} \tag{2.1a}$$

$$\text{s.t.} \quad \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \tag{2.1b}$$

$$\boldsymbol{x} \in \{0,1\}^n \tag{2.1c}$$

where $\boldsymbol{c} \in \mathbb{R}^n$ is the objective coefficient vector, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is the constraint coefficient matrix, and $\boldsymbol{b} \in \mathbb{R}^m$ is the constraint vector. The data $(\boldsymbol{c}, \boldsymbol{A}, \boldsymbol{b})$ specify an *instance* of the problem. Despite its restrictions, this simple 0-1 integer programming (IP) formulation can model a wide variety of real-world optimization problems and be easily extended to more general MILP formulations.

Combinatorial optimization problems have search spaces that grow exponentially as the number of variables increases. This exponential growth results in most combinatorial optimization problems being NP-hard. Moreover, the data explosion in many domains, such as business, finance, logistics, transportation, and telecommunications, leads to larger and more complicated combinatorial optimization problems. Consequently, there is increasing demand for approaches to solving these challenging large-scale problems.

Existing methods for addressing NP-hard combinatorial optimization problems fall into two primary categories with different priorities. Exact methods seek a provably optimal solution by efficiently exploring the entire solution space. They enumerate candidate solutions and systematically discard candidates by theoretical bounds. In contrast, heuristic methods prioritize feasibility over guaranteeing optimality. They explore only a part of the search space to find a reasonable solution, saving computational effort compared to exact methods. However, heuristics sacrifice the ability to prove the solution's optimality.

In this thesis, we focus on exact approaches based on MILP for solving combinatorial optimization problems. In the following, we briefly outline three well-known frameworks for exact solutions: *branch-and-bound* (Section 2.1.1), *cutting plane* (Section 2.1.2), and *branch-and-cut* (Section 2.1.3).

2.1.1 Branch-and-Bound

Branch-and-bound is a principal algorithm for solving MILPs. As its name suggests, branch-and-bound works by two key procedures: *branching* the feasible region into subsets and *bounding* the objective value of the generated subproblems to prune the enumeration.

Given a combinatorial optimization problem formulated as (IP), our goal is to find an optimal solution $\boldsymbol{x}^*$ in the set $S = \{\boldsymbol{x} \in \{0,1\}^n : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$, called the *feasible region*. A *natural linear relaxation* of S is the set $P_0 = \{\boldsymbol{x} \in [0,1]^n : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$ obtained by relaxing the integrality restriction on the vector $\boldsymbol{x}$. The linear program $\min\{\boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{x} \in P_0\}$ is called *the natural LP relaxation* of

(IP). This linear program can be solved efficiently by many algorithms, such as simplex [25] or interior point [26].

If an optimal solution $\boldsymbol{x}^0$ to the natural LP relaxation is integral, it is also an optimal solution to (IP). In the case where $\boldsymbol{x}^0$ has at least an element with fractional value, branch-and-bound uses a *variable selection* strategy to choose an index $j \in \{1, \ldots, n\}$ such that x_j^0 is fractional and divided S into two subsets by adding bounds on the variable x_j (i.e., $x_j = 0$ or $x_j = 1$). We denote these subsets in the form $S(F_0, F_1)$ where $\langle F_0, F_1 \rangle$ is an ordered pair of disjoint sets of indices whose corresponding variables have been fixed to 0 and 1, respectively. Formally, $S(F_0, F_1) = S \cap \{\boldsymbol{x} \in \mathbb{R}^n : x_i = 0 \; \forall i \in F_0, x_i = 1 \; \forall i \in F_1\}$.

Then, S is partitioned into a list of subsets $\mathcal{S} = \{S(\{j\}, \emptyset), S(\emptyset, \{j\})\}$. For each subset $S(F_0, F_1) \in \mathcal{S}$, let IP$(F_0, F_1)$: $\min\{\boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{x} \in S(F_0, F_1)\}$ be the integer program on $S(F_0, F_1)$. Obviously, an optimal solution of (IP) is the best among the optimal solutions of IP(F_0, F_1); thus, to solve (IP), we solve instead the new subproblems corresponding to the subsets in $\mathcal{S}$. Additionally, the minimum among the optimal values of the natural LP relaxations of IP(F_0, F_1) is a lower bound on the optimal value of IP(F_0, F_1).

Given the integer program IP(F_0, F_1) defined on subset $S(F_0, F_1) \in \mathcal{S}$, we consider its natural LP relaxation:

$$\text{LP}(F_0, F_1) : \min\{\boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{x} \in P(F_0, F_1)\}$$

where $P(F_0, F_1)$ is the natural linear relaxation of $S(F_0, F_1)$. The problem IP(F_0, F_1) will be pruned if one of the following three cases occurs:

i) Pruned by infeasibility. LP(F_0, F_1) is infeasible. Then, IP(F_0, F_1) is also infeasible since $S(F_0, F_1) \subseteq P(F_0, F_1)$.

ii) Pruned by integrality. An optimal solution $\boldsymbol{x}^{\text{LP}}$ to LP(F_0, F_1) is integral. Therefore, $\boldsymbol{x}^{\text{LP}}$ is an optimal solution of IP(F_0, F_1) and $\boldsymbol{c}^T\boldsymbol{x}^{\text{LP}}$ is an upper bound on the optimal value of (IP) since $S(F_0, F_1)$ is a subset of S.

iii) Pruned by bound. The optimal value of LP(F_0, F_1) is larger than or equal to the best-known upper bound. Then, $S(F_0, F_1)$ can not contain a better solution than the incumbent one.

If the problem IP(F_0, F_1) is not pruned, branch-and-bound repeats partitioning this problem into two new subproblems by restricting the range of variables.

To systematically solve and bound subproblems, branch-and-bound constructs an enumeration tree where each node corresponds to an LP relaxation of (IP). The root node is associated with the natural LP relaxation of (IP). Two children of a node correspond to two subproblems of the parent's LP relaxation, obtained by adding new constraints on variable ranges. Let $\mathcal{N}$ be the list of active nodes (i.e., that have yet to be pruned or branched on). Algorithm 1 gives a formal description of branch-and-bound.

Algorithm 1 Branch-and-bound algorithm

Input: $(\boldsymbol{c}, \boldsymbol{A}, \boldsymbol{b})$

Output: An optimal solution $\boldsymbol{x}^*$ (if any).

1. *Initialization.*

   Set $\mathcal{N} \leftarrow \{\langle F_0 = \emptyset, F_1 = \emptyset \rangle\}$, $\boldsymbol{x}^* \leftarrow$ `Nil` and `UB` $\leftarrow +\infty$.

2. *Node selection.*

   if $\mathcal{N}$ is empty **then**

   Return $\boldsymbol{x}^*$ and terminate

   else

   Select and remove a node $\langle F_0, F_1 \rangle$ from $\mathcal{N}$

   end if

3. *Prune.*

   Solve LP(F_0, F_1).

   if LP(F_0, F_1) is infeasible **then** go to step 2 ▷ *Pruned by infeasibility*

   else

   Let $\boldsymbol{x}^{\text{LP}}$ be an optimal solution to LP(F_0, F_1).

   if $\boldsymbol{c}^T \boldsymbol{x}^{\text{LP}} \geq$ `UB` **then** go to step 2 ▷ *Pruned by bound*

   else

   if $\boldsymbol{x}^{\text{LP}}$ is integral **then** set `UB` $\leftarrow \boldsymbol{c}^T \boldsymbol{x}^{\text{LP}}$ and go to step 2. ▷ *Pruned by integrality*

   end if

   end if

   end if

4. *Branching.* Pick an index $j \in [n]$ such that $0 < x_j^{\text{LP}} < 1$. Add $\langle F_0 \cup \{j\}, F_1 \rangle$ and $\langle F_0, F_1 \cup \{j\} \rangle$ to $\mathcal{N}$. Go to step 2.

As shown in Algorithm 1, branch-and-bound presents certain open choices that require careful consideration and planning, for example, the node selection criterion in Step 2 and the branching strategy in Step 4. Over the past decades, many approaches to improve selection efficiency have been proposed, but most are hand-crafted heuristics tuned by experts.

2.1.2 The cutting plane method

Another basic algorithm to solve MILP problems is the cutting plane method, which iteratively finds and adds valid inequalities to strengthen the LP relaxation until an optimal integer solution is obtained. The cutting plane method was introduced by Ralph E. Gomory [27] with the use of cutting planes called Gomory cuts.

In particular, consider the IP formulation

$$\text{(IP): } \min\{\boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{x} \in S\}$$

where $S = \{\boldsymbol{x} \in \{0,1\}^n : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$ as the previous section. We first solve the natural LP relaxation of (IP) to obtain an optimal solution $\boldsymbol{x}^0$. As stated earlier, if $\boldsymbol{x}^0$ is integral, then it is an optimal solution to (IP). In the case where $\boldsymbol{x}^0$ does not belong to S, the cutting plane method finds an inequality $\boldsymbol{\alpha}^T\boldsymbol{x} \leq \beta$, $\boldsymbol{\alpha} \in \mathbb{R}^n$, $\beta \in \mathbb{R}$ that is satisfied by all points $\boldsymbol{x}$ in S but violated by $\boldsymbol{x}^0$, namely that $\boldsymbol{\alpha}^T\boldsymbol{x}^0 > \beta$. Such an inequality is called a *cutting plane (cut)* separating $\boldsymbol{x}^0$ from S. The problem of finding cuts is called the *separation problem*.

Let P_0 be the natural linear relaxation of S. Given a cut $\boldsymbol{\alpha}^T\boldsymbol{x} \leq \beta$, denote

$$P_1 = P_0 \cap \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{\alpha}^T\boldsymbol{x} \leq \beta\}.$$

Obviously, $S \subseteq P_1 \subset P_0$. Therefore, the LP relaxation of (IP) on P_1 is *stronger* than the LP relaxation of (IP) on P_0, as it can obtain an equal or better lower bound on the optimal value of (IP). We repeatedly solve LP relaxations and add cuts until an integer solution is obtained.

Formally, let $\mathcal{C}$ be the set of cuts $\boldsymbol{\alpha}^T\boldsymbol{x} \leq \beta$ and $\text{LP}(\mathcal{C})$ be the following LP problem:

$$\begin{aligned}
\text{LP}(\mathcal{C}) : \min\ & \boldsymbol{c}^T\boldsymbol{x} && \text{(2.2a)}\\
\text{s.t.}\ & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} && \text{(2.2b)}\\
& \boldsymbol{\alpha}^T\boldsymbol{x} \leq \beta \quad \forall(\boldsymbol{\alpha}, \beta) \in \mathcal{C} && \text{(2.2c)}\\
& \boldsymbol{x} \in [0,1]^n && \text{(2.2d)}
\end{aligned}$$

Then, the cutting plane method can be illustrated as in Algorithm 2.

Algorithm 2 The cutting plane method

Input: $(\boldsymbol{c}, \boldsymbol{A}, \boldsymbol{b})$

Output: An optimal solution $\boldsymbol{x}^*$ (if any).

```
Set C ← ∅.
while True do
    Solve LP(C) to obtain a solution x^LP.
    if x^LP is integer then
        return x^LP
    else
        Solve the separation problem to find a cut α^T x ≤ β separating x^LP from S.
        C ← C ∪ {(α, β)}.
    end if
end while
```

Central to the cutting plane method is the separation problem. If $\boldsymbol{x}^{\text{LP}}$ is not in S, there are many cuts separating $\boldsymbol{x}^{\text{LP}}$ from S. How do we generate efficient cuts? In general, cuts fall into two categories: *general-purpose cuts*, based on variable integrality, and *combinatorial cuts*, based on problem structure. Each cut type raises distinct design challenges.

In particular, separation procedures for general-purpose cuts are easy to solve and can generate many cuts. However, adding all these cuts to the LP relaxation can significantly increase

its size, making it more challenging to solve. This raises the *cut selection* problem, which chooses high-quality cuts from a candidate set to add to the LP relaxation.

In contrast, separation procedures for combinatorial cuts are more complicated and expensive, but resulting cuts (if any) can significantly tighten the LP relaxation and improve bounds. Thus, we must decide whether to execute these complex separation procedures.

In practice, the cutting plane method is rarely used alone to solve MILP problems because it typically requires an exponential number of cuts to converge to an optimal solution. The cutting plane method also suffers from numerical instability. For these reasons, branch-and-bound was the dominant practical approach for solving MILP problems until Balas et al. (1996) [28] demonstrated that cutting planes could be effectively incorporated within a branch-and-bound framework. This combination enabled cutting planes to contribute substantially to solving MILP problems in practice. This combined approach is known as *branch-and-cut*, which has since replaced branch-and-bound as the leading technique for MILP solvers.

2.1.3 Branch-and-Cut

Branch-and-cut is a combination of two methods: branch-and-bound and cutting plane. Its intuition is to execute separation procedures to find and add several cuts at nodes of the enumeration tree. Adding cuts can tighten bounds for accelerating pruning the enumeration tree. By leveraging the strengths of the two methods, branch-and-cut is the backbone of the state-of-the-art MILP solvers.

Let $\mathrm{LP}(\mathcal{C}, F_0, F_1)$ be the following LP problem:

$$
\begin{aligned}
\min\ & \boldsymbol{c}^T \boldsymbol{x} \\
\text{s.t } & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\
& \boldsymbol{\alpha}^T \boldsymbol{x} \leq \boldsymbol{\beta} && \forall (\boldsymbol{\alpha}, \boldsymbol{\beta}) \in \mathcal{C} \\
& x_i = 0 && \forall i \in F_0 \\
& x_i = 1 && \forall i \in F_1 \\
& \boldsymbol{x} \in [0,1]^n.
\end{aligned}
$$

Algorithm 3 presents a basic branch-and-cut framework.

An additional decision problem of branch-and-cut arises in Step 4 - whether to generate cuts at the current node or proceed directly to branching. This problem is called the *cut generation* problem. In the literature, this decision is often made empirically based on the success and characteristics of previously generated cuts. The size and complexity of the separation problems also influence cut generation decisions. Developing more rigorous and automated cut generation policies remains an open research area.

2.2 Machine Learning

Machine learning is a branch of artificial intelligence that enables computers to learn automatically from data without being explicitly programmed. Using statistical methods, machine learning algorithms build models that progressively improve as they are exposed to more data.

Algorithm 3 Branch-and-cut algorithm

Input: $(\boldsymbol{c}, \boldsymbol{A}, \boldsymbol{b})$

Output: An optimal solution $\boldsymbol{x}^*$ (if any).

1. *Initialization.*

   Set $\mathcal{N} \leftarrow \{\langle F_0 = \emptyset, F_1 = \emptyset \rangle\}$, $\boldsymbol{x}^* \leftarrow$ `Nil` and `UB` $\leftarrow +\infty$.

2. *Node selection.*

   if $\mathcal{N}$ is empty **then**

   Return $\boldsymbol{x}^*$ and terminate

   else

   Select and remove a node $\langle F_0, F_1 \rangle$ from $\mathcal{N}$

   end if

3. *Prune.*

   Solve LP(F_0, F_1).

   if LP(F_0, F_1) is infeasible **then** go to step 2 ▷ *Pruned by infeasibility*

   else

   Let $\boldsymbol{x}^{\text{LP}}$ be an optimal solution to LP(F_0, F_1).

   if $\boldsymbol{c}^T \boldsymbol{x}^{\text{LP}} \geq$ `UB` **then** go to step 2 ▷ *Pruned by bound*

   else

   if $\boldsymbol{x}^{\text{LP}}$ is integral **then** set `UB` $\leftarrow \boldsymbol{c}^T \boldsymbol{x}^{\text{LP}}$ and go to step 2. ▷ *Pruned by integrality*

   end if

   end if

   end if

4. *Branching versus cut generation*

   Decide whether to generate cuts or to branch.

   if generate cuts **then**

   Solve the separation problem and add resulting cuts to $\mathcal{C}$. Go to step 3.

5. *Branching.*

   Pick an index $j \in [n]$ such that $0 < x_j^{\text{LP}} < 1$. Add $\langle F_0 \cup \{j\}, F_1 \rangle$ and $\langle F_0, F_1 \cup \{j\} \rangle$ to $\mathcal{N}$. Go to step 2.

In particular, a machine learning model is trained on a large dataset, called a *training set*, to discover patterns and relationships within the data. The model can then be used to make predictions on new, unseen data. For example, one could train a model on thousands of images of handwritten digits to recognize digits in new images. The ability to adapt and improve based on new data makes machine learning a fast-growing field, especially as computational power and availability of big data continue to expand.

In this section, we present some fundamental notions of two primary machine learning paradigms: *supervised learning* (Section 2.2.1) and *reinforcement learning* (Section 2.2.3). In Section 2.2.2, we introduce neural networks, a branch of machine learning models achieving state-of-the-art results in many machine learning tasks.

2.2.1 Supervised learning

Supervised learning is a machine learning paradigm that trains algorithms by using labeled datasets, where each input sample is labeled by a target vector. The term *"supervised"* refers to the fact that the algorithm learns under the guidance of an expert who knows the correct answers through the labeled data. Supervised learning problems can be categorized as either *classification* or *regression* problems, depending on the desired output. A task is called a classification problem if the outcome is limited to a finite set of discrete categories. Conversely, it is considered a regression problem if the desired output contains at least one continuous variable.

Formally, given a training set $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{N}$ where $\boldsymbol{x}_i$ is an input sample and $\boldsymbol{y}_i$ is its label, we denote respectively by X and Y the input and label spaces. Supervised learning aims to learn a function

$$f_{\mathbf{w}} : X \to Y,$$

where $\mathbf{w}$ is the weights of f, that fits appropriately the training set $\mathcal{D}$. The function $f_{\mathbf{w}}$ is an element of a function space $\mathcal{F}$, called the *hypothesis space*.

To solve a supervised learning problem, we first need to determine how to represent the inputs, as in most practical situations, the input samples have complex structures like text, images, sounds, time series, or graphs. Typically, the raw inputs are transformed into descriptive *feature vectors* through a process called *feature extraction*. Feature extraction is a critical task that requires domain expertise to identify informative features about the inputs. The feature vector should contain enough information for the algorithm to make accurate predictions while maintaining a reasonable dimensionality to avoid the computational burden and the curse of dimensionality. In other words, the features must be descriptive of the underlying patterns in the data but not overly detailed or numerous.

The next step is determining the $f_{\mathbf{w}}$'s form. This selection relies on various factors, such as the properties of the training set, available computing resources, and assumptions about the underlying function. For example, if we have a small dataset but strong assumptions about the target function, parametric models with a fixed number of parameters may be a good choice. On the other hand, non-parametric models, whose number of parameters grows with the training set size, can better handle massive, high-dimensional datasets. However, non-parametric models usually demand substantial computing resources for training.

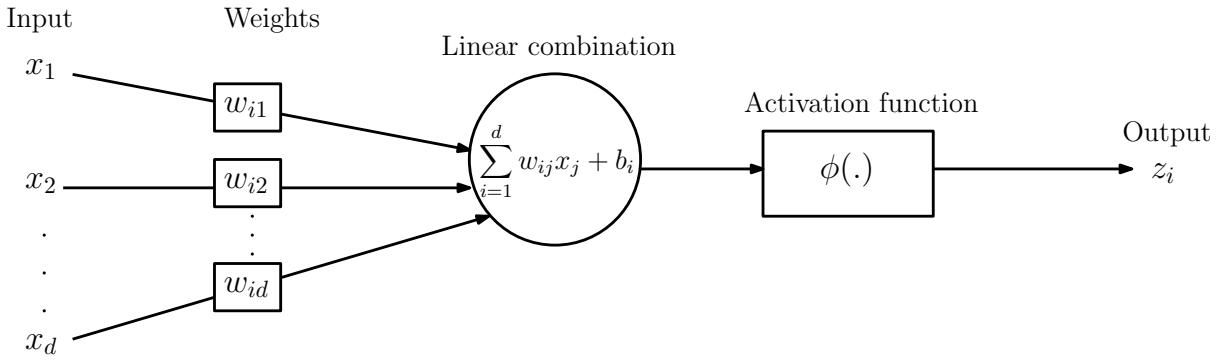


Figure 2.1. The schematic representation of a neuron

Once the input representation and function's form are determined, the weights will be adjusted to make the function $f_{\mathbf{w}}$ appropriately fit the training set. Towards this end, we need to define an *error function* $E_{\mathbf{w}}(\mathcal{D})$ that measures the discrepancy between the samples in the dataset $\mathcal{D}$ and the estimations of $f_{\mathbf{w}}$. A common choice for the error function is the mean squared error between the predictions and target values, i.e.,

$$E_{\mathbf{w}}(\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} \|f_{\mathbf{w}}(\boldsymbol{x}_i) - \boldsymbol{y}_i\|_2^2 \tag{2.4}$$

Notice that the final goal when training a model is to predict new data correctly, not just the data it was trained on. Thus, to evaluate a trained model, we must check both its training error and the gap between training and testing errors. If the training error is too high, the model is *underfitted*. Conversely, if the gap between training and testing error is too large, the model is *overfitted*.

2.2.2 Neural networks

In this section, we present a common choice for the estimated function form: *neural networks*. In recent years, neural networks have achieved state-of-the-art results in various domains like computer vision, natural language processing, speech recognition, and recommendation systems. Their strength lies in their ability to adapt to diverse tasks or data structures, handle large and high-dimensional data sets, and represent highly complex functions. Notably, they can automatically learn valuable features from raw data without manual feature engineering, which traditional models typically require.

2.2.2.1 Multilayer perceptron

A neural network is a sophisticated artificial system that aims to replicate the intricate functions of the human brain. The fundamental elements of the neural network are *processing elements*, which can also be called units, nodes, or neurons. Each neuron receives input from external sources or surrounding units and computes an output signal to transmit to other neurons. Figure 2.1 illustrates a neuron. Formally, a neuron is a function f_i that takes a $d-$dimensional vector $\boldsymbol{x} = (x_1, \ldots, x_d)$ as input, computes a linear combination of $\boldsymbol{x}$'s elements using a weight vector $\boldsymbol{w}_i = (w_{i1}, \ldots, w_{id})$, adds a bias b_i, and applies a non-linear function $\phi(.)$ to yield a final

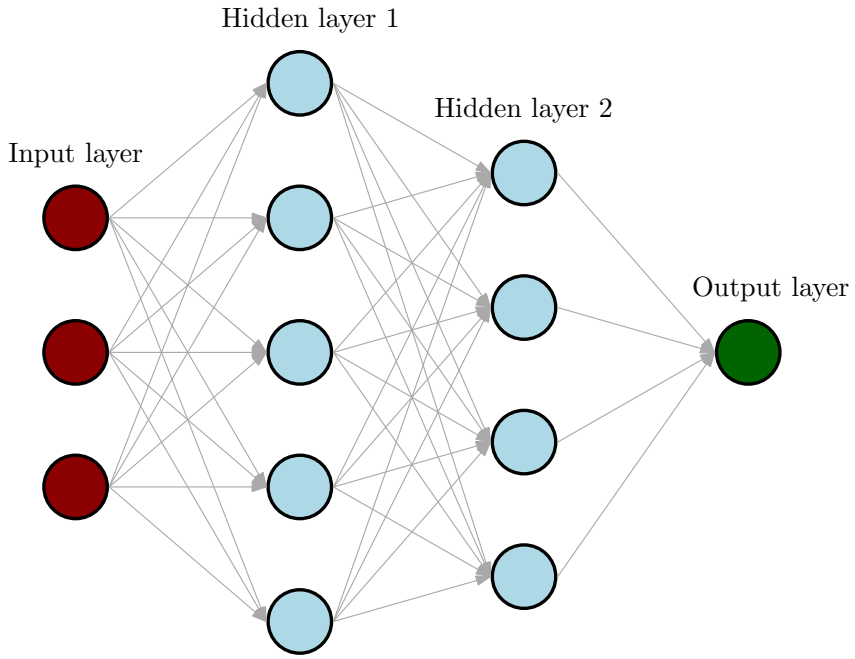


Figure 2.2. A multilayer perceptron with two hidden layers

output z_i. A neuron can be formulated in the following form:

$$z_i = f_i(\boldsymbol{x}) = \phi\left(\sum_{j=1}^{d} w_{ij}x_j + b_i\right).$$

The function $\phi(.)$ is called an *activation function*. This function plays a crucial role in neural networks by introducing non-linearity. The nonlinear aspect enables neural networks to estimate complex functions that would be impossible with a simple linear model. Among the most common activation functions are:

- The sigmoid function:
$$\sigma(x) = \frac{1}{1+\exp(-x)}.$$
- The tanh function:
$$\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}.$$
- The Rectified Linear Unit (ReLU):
$$\text{ReLU}(x) = \max(0, x).$$

A neural network contains a set of neurons organized topologically. One of the most popular types of neural networks is the multilayer perceptron (MLP), which organizes neurons in a feed-forward architecture. In this architecture, neurons are grouped into input, hidden, and output layers. Information flows from one layer to the next and can not be transmitted in the reverse direction. The size of a layer is determined by the number of neurons it contains. Figure 2.2 represents an MLP with two hidden layers.

2.2.2.2 Training neural networks

After constructing a neural network and defining an error function, the next step is to employ a training algorithm to find the network's parameters that minimize the error function. Most training algorithms iteratively adjust the weights by a two-stage procedure: the first stage computes the error function's derivatives with respect to the weights, and the second stage uses these derivatives to update the weights.

The most used and efficient algorithm to compute the derivatives in neural networks is *the back-propagation algorithm* [29], which propagates errors backward through the network. The algorithm's intuition is that the derivatives of the error function in one layer are reused to calculate the derivatives of the error function in the previous layer.

To illustrate, we present the back-propagation computations in a fully-connected MLP model. We consider an MLP with l hidden layers and a nonlinear activation function ϕ identifying for every unit. For simplicity, we only use a single input sample $\boldsymbol{x} = \{x_1, \ldots, x_d\}$ and assume that the target output y is a scalar, so the output layer contains only one unit. Let s_k be the number of units in hidden layer $k \in \{1, \ldots, l\}$ and $s_0 = d$.

In layer k, each unit i first calculates a weighted sum of its inputs, i.e.

$$a_i^k = \sum_{j=1}^{s_{k-1}} w_{ji}^k z_j^{k-1} + b_i^k \tag{2.5}$$

where w_{ji}^k is the weight of node j in layer $k-1$ for incoming node i in the layer k, z_j^{k-1} is the output of node j in layer $k-1$ (note that $z_j^0 = x_j \ \forall j \in \{1, \ldots, d\}$), and b_i^k is the bias of node i in layer k. To further simplify the mathematics, we include b_i^k in the sum by using an additional unit 0 with a fixed output $z_0^{k-1} = 1$. Therefore, (2.5) can be rewritten as

$$a_i^k = \sum_{j=0}^{s_{k-1}} w_{ji}^k z_j^{k-1} \tag{2.6}$$

Then, the activation ϕ is applied to produce the output z_i^k of unit i, i.e.,

$$z_i^k = \phi(a_i^k) \tag{2.7}$$

This process is repeatedly performed from the input layer to the output layer. It is called *forward propagation* since the information is propagated forward through the network. The network returns z_1^l (the output of the unique unit of the output layer) as the predicted output $\hat{y}$. The error function is then computed by:

$$E(\mathbf{w}) = L(\hat{y}, y) \tag{2.8}$$

where L is a loss function that calculate the difference between $\hat{y}$ and y.

Now, we need to compute the derivative of the error function E with respect to a weight w_{ji}^k. Using the chain rule for partial derivative, we have:

$$\frac{\partial E}{\partial w_{ji}^k} = \frac{\partial E}{\partial a_i^k} \cdot \frac{\partial a_i^k}{\partial w_{ji}^k} = \frac{\partial E}{\partial a_i^k} \cdot \frac{\partial}{\partial w_{ji}^k} \left(\sum_{j=0}^{s_{k-1}} w_{ji}^k z_j^{k-1} \right) = \frac{\partial E}{\partial a_i^k} \cdot z_j^{k-1}. \tag{2.9}$$

If we denote

$$\delta_i^k \equiv \frac{\partial E}{\partial a_i^k} \tag{2.10}$$

then (2.9) becomes

$$\frac{\partial E}{\partial w_{ji}^k} = \delta_i^k \cdot z_j^{k-1}. \tag{2.11}$$

The δ_i^k's value is usually called *error*. From (2.11), to compute the derivatives in the network, we only need to calculate errors δ for all units in the hidden and output layers by the following formulations.

- For the output layer:

$$\delta_1^l \equiv \frac{\partial E}{\partial a_1^l} = \frac{\partial E}{\partial z_1^l} \cdot \frac{\partial z_1^l}{\partial a_1^l} = L'(z_1^l) \cdot \phi'(a_1^l) \tag{2.12}$$

- For the hidden layers:

$$\delta_i^k \equiv \frac{\partial E}{\partial a_i^k} = \sum_{j=1}^{s_{k+1}} \frac{\partial E}{\partial a_j^{k+1}} \cdot \frac{\partial a_j^{k+1}}{\partial a_i^k} = \sum_{j=1}^{s_{k+1}} \delta_j^{k+1} \cdot w_{ij}^{k+1} \cdot \phi'(a_i^k) = \phi'(a_i^k) \sum_{j=1}^{s_{k+1}} \delta_j^{k+1} \cdot w_{ij}^{k+1} \tag{2.13}$$

Once the derivatives of the error function with respect to the weights are computed, the next stage is to modify the weights in order to minimize the error function. Most algorithms to tackle this task are based on the Stochastic Gradient Descent (SGD) [30], which updates the weights by:

$$\mathbf{w} \leftarrow \mathbf{w} - \gamma \cdot \frac{1}{m} \cdot \sum_{i=1}^{m} \nabla_{\mathbf{w}} L(f_{\mathbf{w}}(\boldsymbol{x}_i), \boldsymbol{y}_i) \tag{2.14}$$

where γ is the learning rate, $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^m$ is a minibatch sampled from the training set. Intuitively, at each iteration, SGD randomly selects m samples from the training set, computes the gradient, and uses it to update the weights. Algorithm 4 gives a description of SGD.

Algorithm 4 Stochastic Gradient Descent (SGD)

Input: A training set $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$, initial weights $\mathbf{w}$, a learning rate γ
Input: A batch size m, the number of epochs N_e

```
for k ∈ {1, ..., N_e} do
    for l ∈ {1, ..., ⌊N/m⌋} do
        Sample a minibatch {(x_i, y_i)}_{i=1}^m from D.
        Compute the gradient: g ← (1/m) · Σ_{i=1}^m ∇_w L(f_w(x_i), y_i).
        Update w: w ← w − γg.
    end for
end for
```

2.2.2.3 Universal approximation theorem

Neural networks are known for their capacity to approximate a broad range of complex functions. This capacity is guaranteed by the universal approximation theorem proposed by Hornik in 1991 [31]. The theorem proved that a neural network with at least one hidden layer and a

bounded, continuous, and non-decreasing activation function can approximate any continuous function defined on a compact set of $\mathbb{R}^d$. More precisely, the universal approximation theorem can be stated as follows:

Theorem 2.2.1. *Let $\phi : \mathbb{R} \to \mathbb{R}$ be a bounded, continuous, and non-decreasing function and $\mathcal{C}(X)$ be the set of continuous functions on a compact set $X \subset \mathbb{R}^d$, $f \in \mathcal{C}(K_d)$. Then for any $\epsilon > 0$, there exist $\boldsymbol{d} = (d_i)_{i=1,\dots,N} \in \mathbb{R}^N$, $\boldsymbol{b} = (b_i)_{i\in 1,\dots,N} \in \mathbb{R}^N$, and $\boldsymbol{w} \in \mathbb{R}^d$ such that the function $F(\boldsymbol{x})$ defined by*

$$F(\boldsymbol{x}) = \sum_{i=1}^{N} d_i \phi \left(\sum_{j=1}^{d} w_j x_j + b_i \right)$$

satisfies

$$\forall \boldsymbol{x} \in \mathbb{R}^d, |F(\boldsymbol{x}) - f(\boldsymbol{x})| < \epsilon.$$

While the algorithm states that we can approximate any function by a sufficiently large MLP with a single layer, there is no guarantee that the training algorithm can find such a model in practice because of many challenges. For example, the hidden layer may be substantial since the theorem does not provide any upper bound on the number of neurons required. Furthermore, even if the neuron count is reasonable, the training algorithm may fail to find the correct set of parameters corresponding to the desired function or choose inaccurate parameters that overfit the training data.

2.2.2.4 Graph neural networks

As shown in Figure 2.2, the architecture of MLPs does not have mechanisms to treat the relations between the elements of the input vector. When dealing with tasks of graph-structured data, this architecture ignores the relational information inherent in graphs. A more suitable class of neural networks for addressing graph-based tasks is Graph Neural Networks (GNNs). The highlights of GNNs lie in their ability to model complex relationships, their flexibility in handling graph-based tasks and graph variability, and their permutation invariance. The key design element of GNNs is the concept of *message-passing* (MP) that allows GNNs to propagate information through a graph to learn meaningful representations of nodes. In particular, MP layers iteratively update the representations of graph nodes through their neighbors.

Let $G = (V, E)$ be a graph, $\boldsymbol{x}_u$ be the features associated with node $u \in V$, and $\boldsymbol{e}_{uv}$ be the features of edge $uv \in E$. Denote $N(u)$ the set of neighbors of node u in G. An MP layer updates the representation h_u of node u by

$$h_u = \phi \left(\boldsymbol{x}_u, \bigoplus_{v \in N(u)} \psi(\boldsymbol{x}_u, \boldsymbol{x}_v, \boldsymbol{e}_{uv}) \right) \tag{2.15}$$

where $\bigoplus$ is a permutation-invariant aggregation function (e.g., sum, average, and max), ϕ and ψ are differentiable and parameterized message functions (e.g., MLP).

In a GNN, it is common to stack multiple MP layers to increase the model's expressiveness and its ability to capture the local and global context of nodes. However, the number of ML layers used should be chosen carefully. A GNN with an excessive number of MP layers can cause the over-smoothing phenomenon, wherein nodes' representations become indistinguishable.

The outputs of MP layers are the representations of nodes. One can use these representations directly for node-level tasks that perform computation or prediction for each node in a graph. For graph-level tasks, a pooling layer is needed to aggregate the individual node representations into a fixed-size vector representing the entire graph. This layer is referred to as a global pooling layer, which must be permutation invariant (e.g., sum, max, or mean pooling). A permutation invariant pooling layer allows the model to handle different orderings of the nodes while still producing the same graph-level output.

2.2.3 Reinforcement Learning

Another fundamental machine learning paradigm is reinforcement learning (RL). The differences between RL and supervised learning are that RL trains a model through trial-and-error, and its predictions may have long-term effects. In particular, in RL, an agent repeatedly performs actions to interact with the environment and observes responses from the environment to adjust its behaviors in order to maximize a numerical reward measure. This learning mechanism–learning from interaction–is similar to how humans learn. In recent years, RL has been applied to numerous domains, such as operational research, artificial intelligence, game-playing, and control systems.

2.2.3.1 Markov Decision Processes

An RL system contains an *agent* (the decision maker) and an *environment* (the thing the agent interacts with). An RL problem is usually modeled as a Markov Decision Process (MDP), which is defined as a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, r, \gamma)$ where

- $\mathcal{S}$: A set of possible *states* of the environment;
- $\mathcal{A}$: A set of *actions* which the agent may select at each time step;
- $p(s'|s,a)$: A *state transition probability* with

  $$p(s'|s,a) = \mathbb{P}(s_{t+1} = s'|s_t = s, a_t = a),$$

  namely that it gives a probability that the environment moves to state s' when the agent takes action a in state s;
- $r(s,a)$: A bounded *reward function*;
- $\gamma \in [0,1]$: A *discount factor*.

The agent interacts with the environment in the MDP as illustrated in Figure 2.3. Given an initial state s_0, at each time step t, the agent observes state $s_t \in \mathcal{S}$ and takes an action $a_t \in \mathcal{A}$. As a result of this action, the environment moves to a new state s_{t+1} drawn from the transition probability $p(s_{t+1}|s_t, a_t)$ and gives a reward $r_t = r(s_t, a_t)$. Repeating this process results a sequence of states and actions, which refers to as a *trajectory*, or an *episode*. Let $\tau = (s_0, a_0, s_1, a_1, s_2, a_2, \dots)$ be a trajectory experience by the agent. A trajectory ends when it reaches special states, called *terminal states*, or the MDP's horizon which can be infinite.

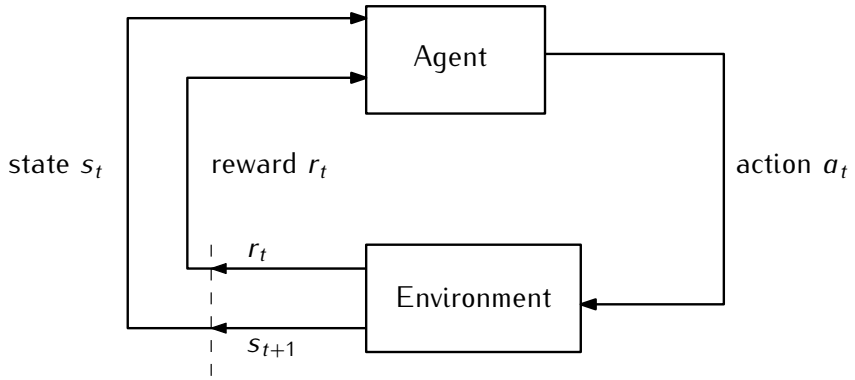


Figure 2.3. An illustration of agent-environment interactions in MDPs at time step t

The return of trajectory τ is the sum of all its collected rewards discounted by γ, i.e.,

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r(s_t, a_t) = r(s_0, a_0) + \gamma r(s_1, a_1) + \gamma^2 r(s_2, a_2) + \dots \tag{2.16}$$

where T can be infinite.

Note that there are various definitions of MDP with slightly different forms of components in the literature. For example, the reward function $r(s, a)$ can be written as a function $r(s, a, s')$ depending on the current state, action, and next state.

2.2.3.2 Value functions and their properties

A *policy* π is a mapping from states to a probability distribution over actions, namely that, $\pi(a|s)$ is the probability the agent takes action a given state s. The aim of RL is to find a policy that maximizes the expected return over trajectories. Toward this end, most RL algorithms involve estimating *value functions*, functions of states that estimate the expected return if the agent starts from a given state and follows its policy thereafter. Formally, the *value function* of state s under policy π is defined by

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{i=0}^{T} \gamma^t r(s_t, \pi(s_t)) \middle| s_0 = s \right] \tag{2.17}$$

for all $s \in \mathcal{S}$. The function V^π is called the *state-value function for policy* π.

Similarly, the *action-value function for policy* π, the so-called $Q-$*function*, gives the expected return for starting from state s, taking action a, and following policy π after that. We can formulate Q^π formally by

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{i=0}^{T} \gamma^t r(s_t, \pi(s_t)) \middle| s_0 = s, a_0 = a \right] \tag{2.18}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.

We then define the *optimal state-value function* $V^* : \mathcal{S} \to \mathbb{R}$ that gives the optimal expected return for starting from a state s:

$$V^*(s) = \max_\pi V^\pi(s) \tag{2.19}$$

for all $s \in \mathcal{S}$. Similarly, the *optimal Q-function* can be defined as:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \tag{2.20}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.

Value functions in RL have a fundamental property of satisfying recursive relationships. Regarding the state-value function as an example, the quantities V^π and V^* satisfy the following equations:

$$V^\pi(s) = r(s,\pi(s)) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^\pi(s') \tag{2.21}$$

$$V^*(s) = \max_{a \in \mathcal{A}} r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^*(s') \tag{2.22}$$

These above equations are the Bellman equations which represent a recursive definition of V^π and V^*. Intuitively, equation (2.21) states that the expected return of state s under policy π is the sum of the immediate reward $r(s,a)$ obtained by picking action $a = \pi(s)$ and the discounted expected returns $\gamma V^\pi(s')$ weighted by probability $p(s'|s,a)$ of the successor states s'. Similarly, equation (2.22) expresses that $V^*(s)$ can be obtain by selecting the best action a ($a = \arg\max_{a' \in \mathcal{A}} r(s,a')$) and then behave optimally from afterwards.

The Bellman equations can be rewritten easily in terms of $Q-$functions through the following connections:

$$V^\pi(s) = Q^\pi(s,\pi(s)) \tag{2.23}$$

$$V^*(s) = \max_{a \in \mathcal{A}} Q^*(s,a) \tag{2.24}$$

Then, the Bellman equations for $Q-$functions are

$$Q^\pi(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) Q^\pi(s',\pi(s')) \tag{2.25}$$

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) \max_{a' \in \mathcal{A}} Q^*(s',a') \tag{2.26}$$

From equation (2.26), if Q^* is given, then the agent can easily determine the optimal policy; for any state s, it simply finds an action that minimizes $Q^*(s,a)$. Finding the optimal policy from V^* is slightly more complicated since the agent has to perform a one-step search. Furthermore, it is also a fact that the optimal solution V^* is unique.

2.2.3.3 RL algorithms

We now briefly introduce several standard algorithms to solve MDPs. Firstly, if we know the transition probability $p(s'|s,a)$, we can solve MDPs by dynamic programming algorithms. For example, by using the optimal Bellman equation (2.22), one can obtain the optimal state-value function by iteratively performing the following update:

$$V(s) \leftarrow \max_{a \in \mathcal{A}} r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V(s') \tag{2.27}$$

until it converges. Then, the optimal policy can be yielded from the resulting optimal state-value function. This method is called *value iteration*.

An alternative algorithm, *policy iteration*, find an optimal policy by generating a sequence of policies $\{\pi_0, \pi_1, \pi_2, \dots\}$ with non-decreasing performance, i.e. $V^{\pi_{k+1}} \geq V^{\pi_k}$, until it converges. Each policy π_k is generated by

$$\pi_k(s) \in \arg\max_{a \in \mathcal{A}} \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^{\pi_{k-1}}(s') \right]. \tag{2.28}$$

In the case we do not know the transition probability, we may first attempt to learn it and use the estimated probabilities for running dynamic programming algorithms. Such methods are called *model-based RL algorithms*. Alternatively, one can try to directly learn value functions, without the transition probability. This approach is called *model-free RL*. For instance, the $Q-$learning algorithm updates the $Q-$function by

$$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha \left[r(s,a) + \gamma \max_{a' \in \mathcal{A}} Q(s',a') \right] \tag{2.29}$$

where α is the learning rate. The $Q-$values of pairs (s,a) are stored in the so-called $Q-$table.

It is important to note that the RL methods presented above are only efficient for small MDPs, where the state space is discrete and small enough to store value functions explicitly. When dealing with larger MDPs, these techniques can become intractable. For instance, if a state is defined by an $n-$dimensional continuous vector, we would need to discretize the state space into a finite set of states. However, the number of discretized states grows exponentially with the dimension n. This exponential growth renders simple RL algorithms impractical for large, high-dimensional state spaces.

Thus, various approaches have been proposed to find an approximation for value functions, e.g., the $Q-$function. Instead of using a finite look-up table to store the $Q-$values of each state-action pair, one can embed the state space in a feature vector $\phi(\mathcal{S})$ and then learn a surrogate $Q-$function $Q_\theta : \phi(\mathcal{S}) \times \mathcal{A} \to \mathbb{R}$ whose weights θ are learned through interactions with the environment. This is known as the *fitted $Q-$iteration method* [32]. A popular choice to represent Q_θ is deep neural networks (called $Q-$networks) due to their approximation capability. However, using a nonlinear function as a neural network to approximate the $Q-$function can lead to instability and divergence during RL for several reasons: correlations between observations in a trajectory, minor updates of Q_θ substantially changing the policy and data distribution, and correlations between action values and target values.

One of the most successful methods for addressing the instability issues of nonlinear function approximation in RL is the *Deep Q-Network (DQN)* algorithm [33]. To reduce correlations between sequential observations, DQN utilizes an experience replay buffer $\mathcal{D} = \{e_1, \dots, e_T\}$ to store collected experiences $e_t = (s_t, a_t, s_{t+1}, r_t)$. During learning, batches B are uniformly sampled from $\mathcal{D}$ to update the $Q-$network weights to minimize the following loss function

$$L_B(\theta) = \mathbb{E}_{(s_i, a_i, s_{i+1}, r_i) \sim \mathcal{D}} \left[\left(Q_\theta(s_i, a_i) - r_i - \gamma \max_{a'} Q_{\bar{\theta}}(s_{i+1}, a') \right)^2 \right] \tag{2.30}$$

where $\bar{\theta}$ are the weights of a separate target network. This target network has an identical architecture to the $Q-$network but is periodically updated with the $Q-$network weights every K steps. Keeping the target network fixed between individual updates helps decrease the correlation between action values $Q(s,a)$ and target values $r + \gamma \max_{a'} Q(s',a')$.

Algorithm 5 DQN algorithm

Initialize a replay buffer $\mathcal{D}$
Initialize a Q-network Q_θ and set $\overline{\theta} = \theta$
for $i = 1, \ldots, n$ **do**
    Set $t = 0$ and an initial state s_0
    while s_t^i is not terminal state **do** ▷ *Execute a trajectory*
        Select a random action a_t^i with probability ϵ;
        otherwise, select $a_t^i = \arg\max_a Q_\theta(s_t^i, a)$ ▷ $\epsilon-$*greedy exploitation*
        Observe next state s_{t+1}^i, and reward $r_t^i = r(s_t^i, a_t^i)$
        Store transition $(s_t^i, a_t^i, s_{t+1}^i, r_t^i)$ into the replay buffer $\mathcal{D}$ ▷ *Experience replay*
        Sample a mini-batch B from $\mathcal{D}$ ▷ *Batch learning*
        Update θ following the gradient derived from equation (2.30) using B
        Every K steps, set $\overline{\theta} \leftarrow \theta$ ▷ *Target network*
        Set $t \leftarrow t + 1$
    end while
end for

When learning through environmental interactions, the agent faces the *exploration-exploitation dilemma* [34]. On the one hand, the agent's knowledge is partial, and thus, it needs to try new actions to gather more information for long-term gains. On the other hand, the agent also wants to yield more immediate rewards by making the best decision based on current information. A common approach to balance this trade-off is $\epsilon-$*greedy exploration*, which randomly selects an action with probability ϵ. Typically, the exploration probability ϵ decreases over time to focus more on exploitation as the agent's knowledge improves. Algorithm 5 gives a formal description of the DQN algorithm.

Part I

Algorithms for fair combinatorial optimization

Chapter 3

Special-purpose branch-and-cut for balanced combinatorial optimization

Summary

In this chapter, we study balanced combinatorial optimization [7], which minimizes the max-min distance of the solution. This problem arises from many real-life situations. For example, a company wants to install a manufacturing line and must source components from multiple suppliers. Let c_{ij} represent the expected lifetime of component j from supplier i. To minimize production downtime for maintenance, the company aims to select components with similar lifespans so that they can be replaced simultaneously. Specifically, they want to choose parts that minimize the range of values for c_{ij} across all selected components.

Another situation relates to the scheduling problem in factories. Let c_{ij} represent the amount of work worker i would need to do if assigned to job j. In order to promote employee satisfaction, the work schedule should minimize the difference between the workload assigned to the most heavily loaded worker and that of the least loaded worker. This balancing of workloads aims to prevent overburdening some workers while underutilizing others.

In these problems, the considered components have an associated cost. The goal is to seek a feasible collection of components for which the difference in cost between the most and least expensive components selected is minimized. Formally, given a finite set E with cost vector $\boldsymbol{c} = \{c_e : e \in E\}$ and a family $\mathcal{F}$ of feasible subsets of E, balanced combinatorial optimization seeks a feasible subset $S^* \in \mathcal{F}$ that minimizes the max-min distance over all $S \in \mathcal{F}$, i.e.,

$$\min_{S \in \mathcal{F}} \left\{ \max_{e \in S} c_e - \min_{e \in S} c_e \right\}.$$

Balanced combinatorial optimization was proposed by Martello et al. [7] with applications related to the assignment problem. In the context of the assignment problem, the set E is the cell set of an $n \times n$ assignment matrix, c_e is the value of cell e, and $\mathcal{F}$ consists of all subsets of cells that constitute assignments. Since its initial formulation, balanced combinatorial optimization has been widely studied, with researchers examining specific cases such as the balanced shortest path [35, 36], the balanced minimum cut [11], the balanced spanning tree [10, 15] and the balanced network flow [37]. Another research direction explores generalizations and variants of balanced combinatorial optimization, as seen in works [14–19, 36, 38–40].

A general framework for solving balanced combinatorial optimization problems is the double-threshold algorithm proposed by Martello et al. [7]. The intuition behind this framework is to find the shortest closed interval such that a subset of E, whose element costs fall within this interval, contains a feasible solution. Such an interval can be found by repeatedly verifying whether a feasible solution exists in a subset of E. Thus, the time complexity of this framework depends on the complexity of the feasibility verification problem. Another approach is to leverage the properties of specific problems to find an optimal solution directly (see Section 3.1 for details). However, this approach has mainly focused on problems that can be solved in polynomial time.

To the best of our knowledge, no exact general MILP framework exists despite MILP's success on many combinatorial optimization problems. The reason is that the balanced optimization objective depends only on the largest and smallest component values. Thus, general-purpose branch-and-cut algorithms without mechanisms to locate the largest and smallest components can not solve balanced combinatorial optimization efficiently. Motivated by this gap, in this chapter, we propose a dedicated branch-and-cut algorithm for balanced combinatorial optimization. We showcase the effectiveness of this algorithm by applying it to the BTSP,

which is an NP-hard problem.

3.1 Literature review

Most algorithms for solving balanced optimization problems are based on two approaches: one is to directly identify a feasible subset $S^* \in \mathcal{F}$ with the smallest max-min distance, and one is to find the minimum range of costs such that the subset $E' \subseteq E$ whose element costs belong to this range contains a feasible subset $S \in \mathcal{F}$. Below, we present some representative works of each approach.

An example of the first approach can be observed in a work of Camerini, Maffioli, Martello, and Toth [9] for the balanced spanning tree problem. Their algorithm starts by arranging edges in ascending order based on their costs and follows by initializing the tree $T = \emptyset$. At each iteration, the edge with the lowest cost that has not been considered is added to T. If T has a cycle, an edge with the smallest cost in the cycle will be removed. The algorithm terminates when the max-min distance equals 0 or all edges are visited. The complexity of this algorithm is $O(mn)$ where m and n, respectively, are the number of edges and nodes of the given graph. Galil and Schieber [10] reduced the algorithm's complexity to $O(m \log n)$ for the undirected case by using dynamic trees of Sleator and Tarjan [41].

In the context of the balanced minimum cut problem, Katoh and Iwano [11] found the minimum range cut by the concept of *upper-critical cuts*. An upper-critical cut with respect to edge e [11] is one whose max-min distance is minimum among all cuts containing e as the smallest edge cost. They proved that the minimum range cut is an upper-critical cut with respect to an edge in the minimum spanning tree. Such a cut can be constructed through the maximum spanning tree and dynamic trees [41]. This approach can solve the problem in $O(m + n \log n)$ time due to the time spent to compute the minimum and maximum spanning trees.

Ahuja [12] proposed a parametric simplex algorithm to find an optimal solution of the balanced linear programming problem, which aims to minimize the difference between the largest cost $c_i x_i$ and the smallest cost $c_j x_j$ given a set of constraints ($\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{x} \geq 0$) and costs for each x_i. In particular, they developed a simplex-based algorithm that implicitly considers the additional constraints representing the largest and smallest costs and operates on the original constraint matrix $\boldsymbol{A}$ to benefit from its special structure.

The second approach to solving balanced optimization problems is based on *the double-threshold algorithm*, a general framework proposed by Martello, Pulleyblank, Toth, and Werra [7]. The intuition of this approach is to reduce balanced optimization problems to the problems of finding the shortest closed interval such that the subset $E' \in E$ whose all element costs lie in this interval contains a feasible subset $S \in \mathcal{H}$. In particular, let $v_1 < v_2 < \cdots < v_k$ be the sorted distinct values of given cost vector $\boldsymbol{c}$. A *feasibility subroutine* takes as input a subset $E' \subseteq E$ and either outputs a $S \in \mathcal{F}$ such that $S \subseteq E'$ or proves that no such S exists. The double-threshold algorithm starts with initializing $l = u = 1$ and $T = \infty$ (T is the smallest max-min distance found so far). Then, it iteratively does the following: first, perform a feasibility subroutine with $E(l, u) = \{e \in E : c_l \leq c_e \leq c_u\}$; if there exists a feasible subset $S \in \mathcal{F}$, update T by $\min\{T, v_u - v_l\}$ and increase l by 1; otherwise, increase u by 1. In total, the algorithm requires

$O(k \cdot f(|E|))$ time where $O(f(|E|))$ denotes the time required to solve a feasible subroutine.

A representative work using this approach is a study of Martello, Pulleyblank, Toth, and Werra [7] for the balanced assignment problem. Here, the feasibility subroutine, which decides whether there exists an assignment in a prescribed subset $E' \subseteq E$, can be implemented in $O(n^{4.5})$ [42]. Thus, the direct application of the double-threshold algorithm would give an $O(n^{4.5})$ algorithm. To reduce the time required for the feasibility subroutine, Martello and his team proposed a modified Hungarian algorithm [43] that can construct an assignment from a partial one. This modification improved the time bound to $O(n^4)$ for the balanced assignment problem.

Larusic and Punnen [13] also adopted the double-threshold framework to solve the BTSP. However, a critical issue of this problem is that its feasibility subroutine is the Hamiltonicity verification problem, which is NP-hard. It causes the approach to be unpractical when the problem instance's size is large. To tackle this issue, Larusic and Punnen heuristically solved the Hamiltonicity verification problem at every iteration. Moreover, they developed four variants of the double-threshold algorithm to reduce the number of iterations without sacrificing solution quality by using the bottleneck TSP [44] and the maximum scatter TSP [45]. With these modifications, their algorithms produced good-quality solutions within 10% optimality, estimated based on lower bounds, to 65 TSPLIB instances [46]. Furthermore, 27 solutions were provably optimal.

3.2 MILP formulation for balanced combinatorial optimization

We consider a combinatorial optimization problem (e.g., the minimum cut, spanning tree, or traveling salesman problem), whose feasible solutions $\mathcal{X} \subseteq \{0,1\}^n$ can be expressed as follows:

$$\begin{aligned} \boldsymbol{A}\boldsymbol{x} &\leq \boldsymbol{b} \\ \boldsymbol{x} &\in \{0,1\}^n \end{aligned}$$

where $\boldsymbol{x}$ is the vector containing decision variables, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\boldsymbol{b} \in \mathbb{R}^m$, n and m are positive integers.

For each $i \in [n]$, let $c_i \in \mathbb{R}$ be the cost of setting x_i to 1. The *balanced problem* defined on a combinatorial set $\mathcal{X}$ aims to minimize the difference between the largest and smallest costs of selected variables, i.e.,

$$\min_{\boldsymbol{x} \in \mathcal{X}} \left\{ \max_{\substack{i \in [n] \\ x_i \neq 0}} c_i x_i - \min_{\substack{i \in [n] \\ x_i \neq 0}} c_i x_i \right\}$$

To linearize this objective function, we denote u and l, respectively, variables representing the highest and lowest component costs. We propose an MILP formulation for balanced combinatorial optimization as follows:

$$\begin{aligned} \text{(BCO)} \quad \min \quad & u - l && && \text{(3.1a)} \\ \text{s.t.} \quad & u \geq c_i x_i && \forall i \in [n] && \text{(3.1b)} \\ & l \leq c_i x_i + (1 - x_i) M && \forall i \in [n] && \text{(3.1c)} \\ & \boldsymbol{x} \in \mathcal{X} && && \text{(3.1d)} \end{aligned}$$

where M is a constant such that $l \leq M$ (e.g., $M = \max_{i\in[n]} c_i$). Constraints (3.1b) and (3.1c) are used to estimate the highest and lowest costs. More specifically, constraints (3.1b) ensure that u must be greater than or equal to the costs of all components selected in the solution. On the other hand, if a variable x_i is set to 1, inequalities (3.1c) read as $l \leq c_i$, which are true by the definition of l. Otherwise (i.e., $x_i = 0$), constraints (3.1c) become $l \leq M$, which are valid by the definition of M.

The choice of M in constraints (3.1c) depends on the characters of the combinatorial set $\mathcal{X}$. M can be chosen globally for all variables or specifically for each variable x_i, $i \in [n]$. Generally, when the value of M decreases, the bounds of l become tighter. For illustration, we present an instantiation of our general model (BCO) on the TSP.

Example 3.2.1. Given an undirected graph $G = (V, E)$ with edge costs $\boldsymbol{c}$, the BTSP consists in finding a tour that minimizes the max-min distance, i.e.,

$$\min_{\mathcal{H}\in\Pi(G)} \left\{ \max_{e\in\mathcal{H}} c_e - \min_{e\in\mathcal{H}c_e} \right\}$$

where $\Pi(G)$ is the set of all Hamiltonian cycles in G. The BTSP is NP-hard by folklore that the problem of finding a Hamiltonian cycle in the graph can be reduced to a BTSP with unit edge costs.

We denote by $\{x_e \mid e \in E\}$ a set of binary variables where $x_e = 1$ if edge e is in the tour and $x_e = 0$ otherwise. Let u and l, respectively, be variables representing the tour's highest and smallest edge costs. We denote by $\delta(S)$ the set of edges that have exactly one end-vertex in $S \subset V$; $\delta(\{v\})$ is abbreviated as $\delta(v)$ for $v \in V$. The BTSP can be written as the following MILP formulation:

$$\begin{aligned}
(\text{BTSP}) \quad \min \quad & u - l && && (3.2a)\\
\text{s.t.} \quad & u \geq c_e x_e && \forall e \in E && (3.2b)\\
& l \leq c_e x_e + (1 - x_e) M_e && \forall e \in E && (3.2c)\\
& \sum_{e\in\delta(v)} x_e = 2 && \forall v \in V && (3.2d)\\
& \sum_{e\in\delta(S)} x_e \geq 2 && \forall \emptyset \neq S \subset V && (3.2e)\\
& x_e \in \{0, 1\} && \forall e \in E && (3.2f)
\end{aligned}$$

where $M_e = \min\{\max_{e'\in\delta(i)} c_{e'}, \max_{e'\in\delta(j)} c_{e'}\}$ for all $e = (i, j) \in E$. As in (BCO), constraints (3.2b) and (3.2c) are used to estimate the largest and smallest edge costs of the tour. Obviously, $l \leq \max_{e\in\delta(i)} c_e, \forall i \in V$ and constraints (3.2c) are thus valid. The combinatorial set $\mathcal{X}$ of the BTSP is represented by the subtour elimination polytope [47]. In particular, constraints (3.2d) are degree constraints, which ensure that each vertex has precisely two incident edges in the tour. Constraints (3.2e) are the well-known subtour elimination constraints (SECs) that prevent the existence of subtours.

3.2.1 Need for a dedicated branch-and-cut algorithm for balanced combinatorial optimization

Obviously, (BCO) can be solved by a general-purpose branch-and-cut algorithm. However, the main challenge of balanced combinatorial optimization is not representing combinatorial set $\mathcal{X}$ but estimating the largest and smallest costs. The lack of mechanisms to address this task causes general-purpose branch-and-cut algorithms can not solve balanced combinatorial optimization efficiently.

This issue can be demonstrated by the following experiment. We solve the TSP and BTSP on the TSPLIB instance si175 (with 175 vertices) [46] by the commercial solver CPLEX 12.10, a widely-used general-purpose branch-and-cut algorithm for MILP problems. We formulate both problems by the same tour's constraints, i.e., degree constraints (3.2d), SECs (3.2e), and integral constraints (3.2f). The CPU time limit is set to 10800 seconds.

Table 3.1 shows the results of the two problems. Column "*Time*" indicates the running time in seconds of the solver. Column "*Gap*" gives the best IP relative gap so far. Columns "*Nodes*" and "*Depth*" respectively report the number of nodes and the depth of the search tree (i.e., the number of edges along the longest path from the root node to a leaf node).

Problem	**Time**	**Gap (%)**	**Nodes**	**Depth**
TSP	25.52	0.00	4324	89
BTSP	10800.09	28.57	140003	987

Table 3.1. The results of the TSP and BTSP on the instance si175.

As shown in Table 3.1, the BTSP can not be solved to optimality (the value of the IP relative gap is 28.57%) within the CPU time limit, while the TSP is solved easily in 25 seconds. The tree size and tree depth of the BTSP are substantially larger than those of the BTSP.

A cause of these results is the use of the big-M in constraints (3.2c) to represent the smallest edge cost l. Constraints (3.2c) force l to be smaller than or equal to c_e if an edge e occurs in the tour $(x_e = 1)$. The big-M is used to guarantee the validity of (3.2c) when an edge e is not a tour's edge $(x_e = 0)$. However, the big-M can lead to loose bounds for l in the LP relaxations. For example, when $M_e = 100, c_e = 10$, and $x_e = 0.1$, constraints (3.2c) allow l to take a value of 91, while l should not exceed 10 as the edge e is used in the solution.

Another reason relates to the objective function of the BTSP. Unlike the TSP whose objective function depends on all edges selected in the tour, the objective function of the BTSP only depends on the values of the largest and smallest edge costs. Unless we change all edges with maximum or minimum cost in the tour, the objective value of the BTSP remains unchanged. Thus, the solver has to branch on many variables to discard subtrees of the branch-and-bound tree. It leads to an enormous search tree, which is time-consuming to explore (as illustrated in Table 3.1).

These issues drive us to design a special-purpose branch-and-cut algorithm for balanced combinatorial optimization, whose central is a new class of local cuts to tighten bounds on the smallest component cost. This cut class will be presented in the next section.

3.3 Local bounding cuts

The experimental results presented in Section 3.2.1 empirically demonstrate that accurately estimating the maximum and minimum component costs in balanced combinatorial optimization impacts the performance of the branch-and-cut method enormously. In (BCO), while the highest cost can be directly estimated using the decision variables, the lowest cost requires a big enough constant M. However, the value of M that validates constraints (3.1c) often provides only loose bounds on the smallest cost in the LP relaxations. To address this, we introduce a family of local cuts, called *local bounding cuts*. These cuts aim to improve the value of M during the solving process and reinforce the bounds of l in the enumeration tree. Importantly, these cuts do not rely on problem-specific structures, making them applicable to any balanced combinatorial optimization problem.

Observe that in the enumeration tree, each node is associated with an ordered pair $\langle F_0, F_1 \rangle$ of two disjoint index sets where F_0, $F_1 \subseteq [n]$ respectively contain the index of variables that have been fixed to 0 and 1. Given a tree node $\langle F_0, F_1 \rangle$, a feasible solution found by this node or its descendants satisfies

$$x_i = 0 \qquad \forall i \in F_0$$
$$x_i = 1 \qquad \forall i \in F_1.$$

Let $I_{C(F_1)}$ be the minimum of $C(F_1) = \{c_i \mid i \in F_1\}$. Obviously, the smallest component cost of this feasible solution can not exceed $I_{C(F_1)}$. Based on this observation, we generate local bounding cuts as follows:

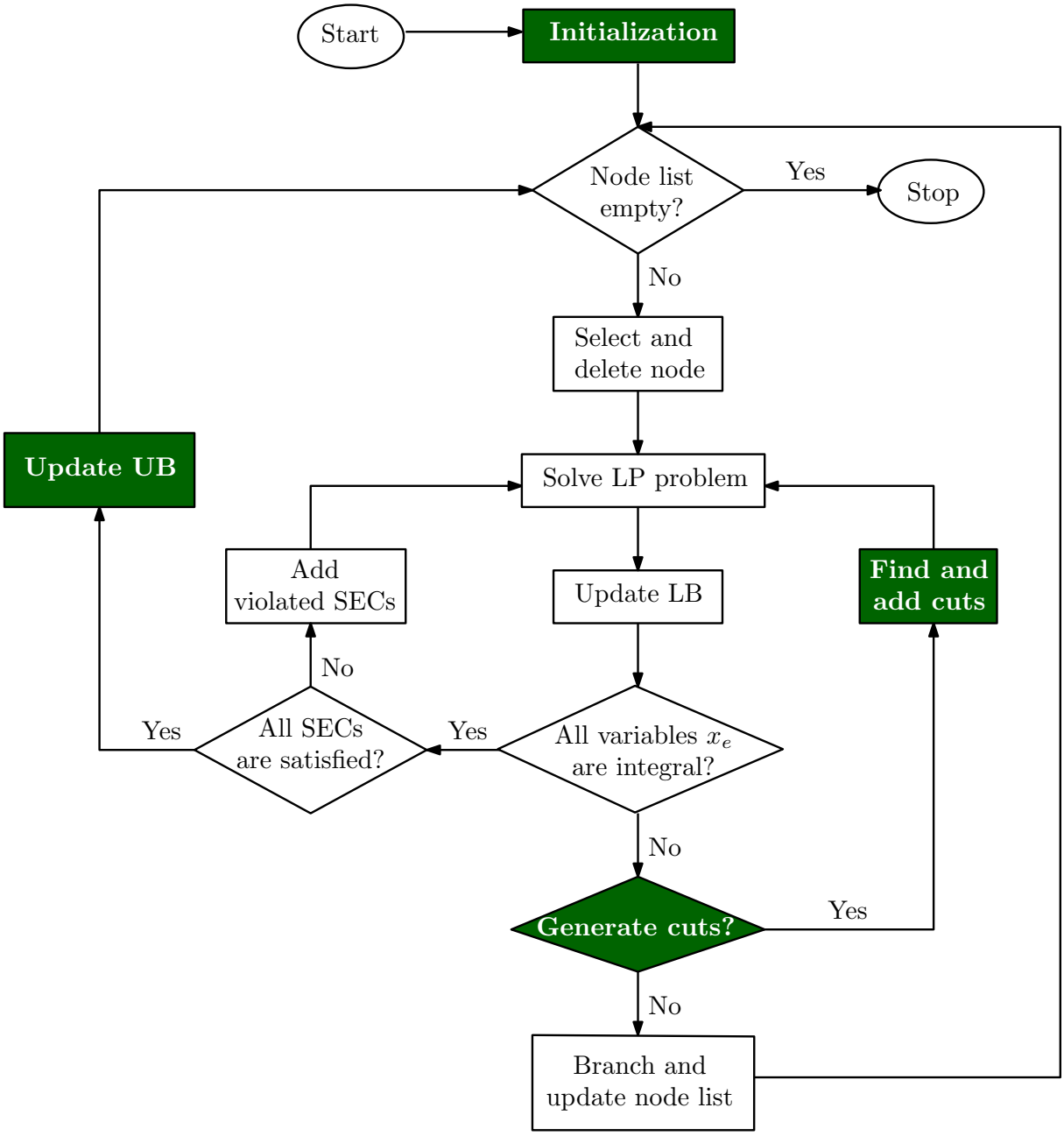
$$l \leq c_i x_i + (1 - x_i) I_{C(F_1)} \quad \forall i \in [n]. \tag{3.3}$$

As their name suggests, the local bounding cuts are locally valid, namely that these cuts are valid only for the current node and its descendants in the enumeration tree, as they use the specific information at the node. The local bounding cuts aim to favor early locating the smallest component cost at the subtree to help the solver concentrate on finding a feasible solution or proving the solution's non-existence. These cuts can tighten the bounds of the smallest cost l in the LP relaxations and thus narrow the interval $[l, u]$.

3.4 Illustration: Branch-and-cut algorithm for the BTSP

In this section, we specialize the proposed branch-and-cut algorithm for the BTSP, an NP-hard case of balanced combinatorial optimization. Besides general-purpose local bounding cuts, we develop problem-specific algorithms to initialize lower and upper bounds on the optimal value of the BTSP and techniques to eliminate and fix variables to improve the algorithm's performance further. We first present the general schema of our algorithm, followed by an in-depth description of its vital components.

Figure 3.1 illustrates the flowchart of our branch-and-cut algorithm for the BTSP. After the initialization step, the algorithm constructs an enumeration tree where each node corresponds to an LP relaxation of (BTSP). The tree's root node is associated with an LP relaxation obtained by dropping all SECs (3.2e) and the integrality requirements on the variables x_e, $e \in E$. At each iteration, we select an active node (that has yet to be pruned or branched on) and solve the corresponding LP relaxation. If we obtain an optimal solution in which all x_e values are integral,



LP: linear programming; LB: lower bound; UB: upper bound; SECs: subtour elimination constraints

Figure 3.1. The schema of our branch-and-cut algorithm for the BTSP. Our improvements focus on the green components in the diagram.

we verify the satisfaction of SECs and add violated ones to the LP relaxations. Otherwise, we can solve separation problems to generate cuts in order to strengthen LP relaxations or select a variable having a fractional value to branch.

Our improvements focus on initializing, enhancing the upper bound, and generating cutting planes on the enumeration tree. Other components of the branch-and-cut algorithm, such as node selection or branching, follow the default rules of the commercial solver CPLEX 12.10.

The tightness of lower and upper bounds on the optimal value of the BTSP is crucial for pruning the enumeration tree. To initialize good bounds, we develop a lower bounding algorithm based on the biconnectivity of Hamiltonian cycles (Section 3.4.1) and a local search algorithm for the BTSP based on $k-$opt algorithms [48, 49] (Section 3.4.2). The bounds found by these algorithms are used not only to provide warm starts for the branch-and-cut algorithm but also to sparsify the instance graph by eliminating edges that can not be part of an optimal solution (Section 3.4.3).

During the branch-and-cut algorithm, to further improve the upper bound, when a new incumbent solution is found, we perform our local search algorithm for the BTSP to enhance this solution. To tighten the lower bound, we use local bounding cuts proposed in Section 3.3 and develop *variable fixing cuts* (Section 3.4.4). Moreover, we also propose strategies to decide whether to generate cuts or to branch at a tree node, which are presented in Section 3.4.5.

A formal description of the proposed branch-and-cut algorithm is presented in Algorithm 6. In the remainder of this section, we present the improved components in detail. For the sake of clarity, we present below some notations used throughout this section.

3.4.0.1 Preliminaries

Given an undirected graph $G = (V, E)$ and a cost vector $\boldsymbol{c}$ associated with E, for any subset S of C, we denote by $\delta(S)$ a subset of E where each edge has exactly one end-vertex in S, i.e., $\delta(S) = \{(i, j) \in E : i \in S \text{ and } j \in V \setminus S\}$. For abbreviation, we write $\delta(v)$ instead of $\delta(\{v\})$ for all $v \in V$. Given a Hamiltonian cycle $\mathcal{H} \in \Pi(G)$, we respectively denote by $u_{\mathcal{H}}$ and $l_{\mathcal{H}}$ the largest and smallest edge costs in $\mathcal{H}$. For an edge set $F \subseteq E$, we denote $V(F)$ the end-vertices set of edges in F and $C(F) = \{c_e \in \boldsymbol{c} \mid e \in F\}$ the edge cost set corresponding to F. Without loss of generality, we assume that $C(E) = \{C_1, \ldots, C_p\}$ where $p \leq m$ is the number of distinct components of the cost vector $\boldsymbol{c}$ and $C_1 < C_2 < \cdots < C_p$. For an interval $[\alpha, \beta]$, $G[\alpha, \beta]$ stands for a subgraph of G with edge set $E[\alpha, \beta] = \{e \in E \mid \alpha \leq c_e \leq \beta\}$. We call $G[\alpha, \beta]$ *the subgraph restricted by* $[\alpha, \beta]$.

3.4.1 Lower bounding algorithm

Given a graph $G = (V, E)$ with edge costs $\boldsymbol{c}$, we present below an algorithm partly inspired by the Hamiltonian verification procedure in [13] to compute a lower bound on the optimal value of the BTSP at the initialization step.

As mentioned in [13], a Hamiltonian graph must be a biconnected graph (i.e., a graph in which for any pair of vertices u and v, there exist two paths from u to v without any vertices in common except u and v). The intuition of the algorithm is that for all distinct costs $C_i \in C(E)$, we find the shortest interval containing C_i such that the subgraph restricted by this interval is

Algorithm 6 Branch-and-cut algorithm for the BTSP

Input: An undirected graph $G = (V, E)$ with cost vector $\boldsymbol{c} \in \mathbb{R}_+^{|E|}$.
Output: A tour whose max-min distance is minimum.

1. *Initialization*

   Find a lower bound $\underline{z}$. ▷ Section 3.4.1

   Find a feasible solution $(\boldsymbol{x}', u', l')$ of (BTSP). Let $\overline{z}$ be an upper bound on the optimal value of the BTSP. ▷ Section 3.4.2

   Eliminate edges that can not occur in an optimal solution. ▷ Section 3.4.3

   Let N_0 be the node corresponding to the LP relaxation of (BTSP) obtained by relaxing all SECs and integrality constraints.

   Set $\overline{z} \leftarrow u' - l'$, $\mathcal{C} \leftarrow \emptyset$, $\mathcal{N} \leftarrow \{\langle F_0 = \emptyset, F_1 = \emptyset\rangle\}$, $(\boldsymbol{x}^*, u^*, l^*) \leftarrow (\boldsymbol{x}', u', l')$.

2. *Node selection.*

   if $\mathcal{N}$ is empty **then**

   Return $(\boldsymbol{x}^*, u^*, l^*)$ and terminate

   else

   Select and remove a node $\langle F_0, F_1\rangle$ from $\mathcal{N}$

   end if

3. **Prune**

   Solve LP$(\mathcal{C}, F_0, F_1)$.

   if LP$(\mathcal{C}, F_0, F_1)$ is infeasible **then** go to step 2

   else

   Let $(\boldsymbol{x}^{\mathrm{LP}}, u^{\mathrm{LP}}, l^{\mathrm{LP}})$ be an optimal solution of LP$(\mathcal{C}, F_0, F_1)$.

   if $u^{\mathrm{LP}} - l^{\mathrm{LP}} \geq \overline{z}$ **then** go to step 1

   else

   if $\boldsymbol{x}^{\mathrm{LP}}$ is integral **then**

   if $\boldsymbol{x}^{\mathrm{LP}}$ satisfies all SECs **then**

   Enhance $(\boldsymbol{x}^{\mathrm{LP}}, u^{\mathrm{LP}}, l^{\mathrm{LP}})$ by the local search algorithm ▷ Section 3.4.2

   Fix variables based on $(\boldsymbol{x}^{\mathrm{LP}}, u^{\mathrm{LP}}, l^{\mathrm{LP}})$. ▷ Section 3.4.4

   Set $(\boldsymbol{x}^*, u^*, l^*) \leftarrow (\boldsymbol{x}^{\mathrm{LP}}, u^{\mathrm{LP}}, l^{\mathrm{LP}})$ and $\overline{z} \leftarrow z^{\mathrm{LP}}$. Go to step 1.

   else

   Add SECs violated by $\boldsymbol{x}^{\mathrm{LP}}$ to $\mathcal{C}$. Go to step 3.

   end if

   end if

4. *Branching versus cut generation*

   Decide whether to generate cuts or to branch. ▷ Section 3.4.5

   if generate cuts **then**

   Solve separation problems and add resulting cuts. ▷ Sections 3.3, 3.4.4

   else

   Go to Step 5

5. *Branching*

   Pick an index $j \in [m]$ such that $0 < x_j^{\mathrm{LP}} < 1$. Add $\langle F_0 \cup \{j\}, F_1\rangle$ and $\langle F_0, F_1 \cup \{j\}\rangle$ to $\mathcal{N}$. Go to step 2.

biconnected. The minimum length among these intervals is a lower bound on the minimum max-min distance of the BTSP. Algorithm 7 gives a formal description of our lower bounding algorithm. Before describing the algorithm in detail, we introduce some definitions and lemmas.

Algorithm 7 Lower bounding algorithm for the BTSP

Input: A graph $G = (V, E)$ with edge costs $\boldsymbol{c}$.
Output: A lower bound on the optimal value of the BTSP.

Let $C_1 < C_2 < \cdots < C_p$ be the distinct costs of $\boldsymbol{c}$
$b_0 \leftarrow 1; b_i \leftarrow p+1, \forall i \in [p]; C_{p+1} \leftarrow +\infty$
for $i \in [p]$ **do**
    $j \leftarrow b_{i-1}$
    while $j \leq p$ **do**
        if $G[C_i, C_j]$ is biconnected **then**
            $b_i \leftarrow j$
            break
        end if
        $j \leftarrow j+1$
    end while
end for
for $i \in [p]$ **do**
    $l_i \leftarrow 1, u_i \leftarrow p$
    for $j \in [i]$ **do**
        if $C_{b_j} - C_j < C_{u_i} - C_{l_i}$ **then**
            $l_i \leftarrow j, u_i \leftarrow b_j$
        end if
    end for
end for
return $\min_{i \in [p]} C_{u_i} - C_{l_i}$.

Definition 3.4.1 (Biconnected interval)**.** For any $C_i \in C(E)$, a *biconnected interval compatible with* C_i is an interval $[\alpha, \beta]$ such that

i) $\alpha \leq C_i \leq \beta$;

ii) $G[\alpha, \beta]$ is biconnected.

The length of a biconnected interval $[\alpha, \beta]$ is the difference between β and α, i.e., $\beta - \alpha$. We denote by $\gamma(C_i)$ the length of the shortest biconnected interval compatible with C_i.

Lemma 3.4.2. *Let $\mathcal{H}$ be a tour in G. If $\mathcal{H}$ contains an edge with cost C_i, then*

$$u_{\mathcal{H}} - l_{\mathcal{H}} \geq \gamma(C_i).$$

Proof. We consider the graph $G[l_{\mathcal{H}}, u_{\mathcal{H}}]$ with edge set $E[l_{\mathcal{H}}, u_{\mathcal{H}}] = \{e \in E \mid l_{\mathcal{H}} \leq c_e \leq u_{\mathcal{H}}\}$. $G[l_{\mathcal{H}}, u_{\mathcal{H}}]$ is biconnected as it contains the tour $\mathcal{H}$. Since $\mathcal{H}$ has an edge with cost C_i, $l_{\mathcal{H}} \leq C_i \leq u_{\mathcal{H}}$. Thus, $(l_{\mathcal{H}}, u_{\mathcal{H}})$ is a biconnected interval compatible with C_i. By the definition of $\gamma(C_i)$, $u_{\mathcal{H}} - l_{\mathcal{H}} \geq \gamma(C_i)$. □

Corollary 3.4.3. *Let $\gamma^* = \min_{C_i \in C(E)} \gamma(C_i)$ and $\mathcal{OPT}$ be the optimal value of (MILP-BTSP), we have $\gamma^* \leq \mathcal{OPT}$.*

Thanks to Corollary 3.4.3, to obtain a lower bound on the optimal value of the BTSP, it is sufficient to find the shortest biconnected interval compatible with C_i for all $C_i \in C(E)$. The following lemma provides a characterization of the shortest biconnected intervals.

Lemma 3.4.4. *If $[\alpha, \beta]$ is the shortest biconnected interval compatible with C_i, then α and β belong to the edge cost set of E.*

Proof. We consider the graph $G[\alpha, \beta]$. Let $\alpha' = \min\{c_e \mid e \in E[\alpha, \beta]\}$ and $\beta' = \max\{c_e \mid e \in E[\alpha, \beta]\}$. Obviously, $\alpha', \beta' \in C(E)$ and $\alpha' \leq C_i \leq \beta'$. Since $G[\alpha', \beta'] = G[\alpha, \beta]$ and $G[\alpha, \beta]$ is biconnected, $G[\alpha', \beta']$ is also biconnected. Thus, $[\alpha', \beta']$ is a biconnected interval compatible with C_i.

Since $[\alpha, \beta]$ is the shortest biconnected interval compatible with C_i, $\beta - \alpha \leq \beta' - \alpha'$. On the other hand, by the definition of $G[\alpha, \beta]$, $\alpha \leq \alpha'$ and $\beta \geq \beta'$. Then, $\beta' - \alpha' \leq \beta - \alpha$. The equality holds if and only if $\alpha = \alpha'$ and $\beta = \beta'$. □

By Lemma 3.4.4, to find the shortest biconnected intervals, we first determine the smallest index $b_j \in [p]$ such that $G[C_j, C_{b_j}]$ is biconnected, for all $C_j \in C(E)$. Note that the b_j computation is according to the increasing ordering of j (i.e., starting with C_1, followed by C_2, and so on). Then, the shortest biconnected interval compatible with C_i is the shortest interval $[C_j, C_{b_j}]$ containing C_i. A naive way to find b_j is to initially set b_j by j and increase b_j until $G[C_j, C_{b_j}]$ is biconnected. It requires checking the graph's biconnectivity $O(|E|^2)$ times. We can reduce it to $O(|E|)$ by using the following lemma.

Lemma 3.4.5. *For any $i, j \in [p]$, if $C_i < C_j$ then $b_i \leq b_j$.*

Proof. We prove the lemma by contradiction. Assume that there exist two costs C_i, C_j such that $C_i < C_j$ and $b_i > b_j$. Obviously, $G[C_j, C_{b_j}]$ is a subgraph of $G[C_i, C_{b_j}]$. Since $G[C_j, C_{b_j}]$ is biconnected, $G[C_i, C_{b_j}]$ is also biconnected. On the other hand, b_i is the smallest value such that $G[C_i, C_{b_i}]$ is biconnected. Thus, $b_i \leq b_j$, contradicts the assumption. □

Using Lemma 3.4.5, we can set b_j initially as b_{j-1} instead of j. This reduces the number of biconnectivity checks at most $O(|E|)$. The algorithm then repeatedly verifies the biconnectivity of the graph $G[C_j, C_{b_j}]$ and increases b_j until $G[C_j, C_{b_j}]$ is a biconnected graph. Since a biconnected graph is a connected graph without articulation vertices, the graph's biconnectivity can be checked in $O(|V| + |E|)$ by Tarjan's algorithm [50]. In total, the complexity of Algorithm 7 is $O(|E|^2)$.

3.4.2 Local search algorithm

We now propose a local search algorithm for the BTSP, called $k-$*balanced*, based on $k-$opt algorithms for the TSP [48, 49]. The algorithm takes a graph $G = (V, E)$ with edge costs $\boldsymbol{c}$, a tour $\mathcal{H}$, and a constant k as input and returns an improved tour with a smaller max-min distance. The purpose of this algorithm is twofold: i) to provide a good upper bound at the initialization step, and ii) to enhance the incumbent solution in the course of the branch-and-cut algorithm.

The intuition of $k-$balanced is to repeatedly perform $k-$exchanges ($k-$opt moves) to improve the current tour. A $k-$exchange replaces k edges in the current tour with k edges in such a way that a tour with a smaller max-min distance is achieved. Algorithm 8 sketches a generic version of $k-$balanced. In the following, we describe in detail the algorithm.

Algorithm 8 Generic k-balanced

Input: A graph $G = (V, E)$ with edge costs $\boldsymbol{c}$, a tour $\mathcal{H}$, and a constant k.
Output: A tour $\mathcal{H}'$ such that $u_{\mathcal{H}'} - l_{\mathcal{H}'} < u_{\mathcal{H}} - l_{\mathcal{H}}$.

```
improved ← True
while improved do
    improved ← False
    Select (F, l', u') where F ⊂ H and [l', u'] ⊊ [l_H, u_H].
    EC(F, l', u') ← {(i, j) ∈ E | i, j ∈ V(F) ∧ (l' ≤ c_(i,j) ≤ u')} .
    if exists a k-subset F̄ ⊂ EC(F, l', u') such that (H \ F) ∪ F̄ is a tour then
        H ← (H \ F) ∪ F̄.
        improved ← True
    end if
end while
return (H, l_H, u_H).
```

Given a tour $\mathcal{H}$ of G, at each iteration, k-balanced constructs two edge sets, $F = \{f_1, \ldots, f_k\}$ and $\overline{F} = \{\overline{f}_1, \ldots, \overline{f}_k\}$, such that $\mathcal{H}' = (\mathcal{H} \setminus F) \cup \overline{F}$ is a new tour with a smaller max-min distance. We call the edges of F *out-edges* and the edges of $\overline{F}$ *in-edges*.

The max-min distance of $\mathcal{H}'$ is smaller than that of $\mathcal{H}$ if and only if all edge costs of $\mathcal{H}'$ belong to an interval shorter than $[l_{\mathcal{H}}, u_{\mathcal{H}}]$. Due to this fact, the out-edge set F must contain all edges with either the maximum edge cost or the minimum edge cost in $\mathcal{H}$ and the in-edge set $\overline{F}$ only comprises edges with costs belonging to a range $[l', u']$ such that $u' - l' < u_{\mathcal{H}} - l_{\mathcal{H}}$. In order to avoid searching all possible intervals $[l', u']$, we simply consider intervals $[l', u'] \subsetneq [l_{\mathcal{H}}, u_{\mathcal{H}}]$.

We first describe a way to construct the in-edge set $\overline{F}$ given a triple (F, l', u') where $F \subset \mathcal{H}$ and $[l', u'] \subsetneq [l_{\mathcal{H}}, u_{\mathcal{H}}]$. Let $EC(F, l', u') = \{(i, j) \in E \mid i, j \in V(F) \wedge (l' \leq c_{(i,j)} \leq u')\}$ the set of edges whose end-vertices are in $V(F)$ with costs between l' and u'. By its definition, $EC(F, l', u')$ is precisely the set of edges that can be used to reconnect a tour from $\mathcal{H} \setminus F$, namely that $\overline{F} \subset EC(F, l', u')$. To construct $\overline{F}$, we solve the problem of completing a Hamiltonian cycle from $\mathcal{H} \setminus F$ with only edges in $EC(F, l', u')$. With k fixed, we can solve the same problem on G' - a compressed version of G with at most $2k$ vertices. The construction of $\overline{F}$ is thus cheap since it is independent of the size of G. Figure 3.2 illustrates this idea.

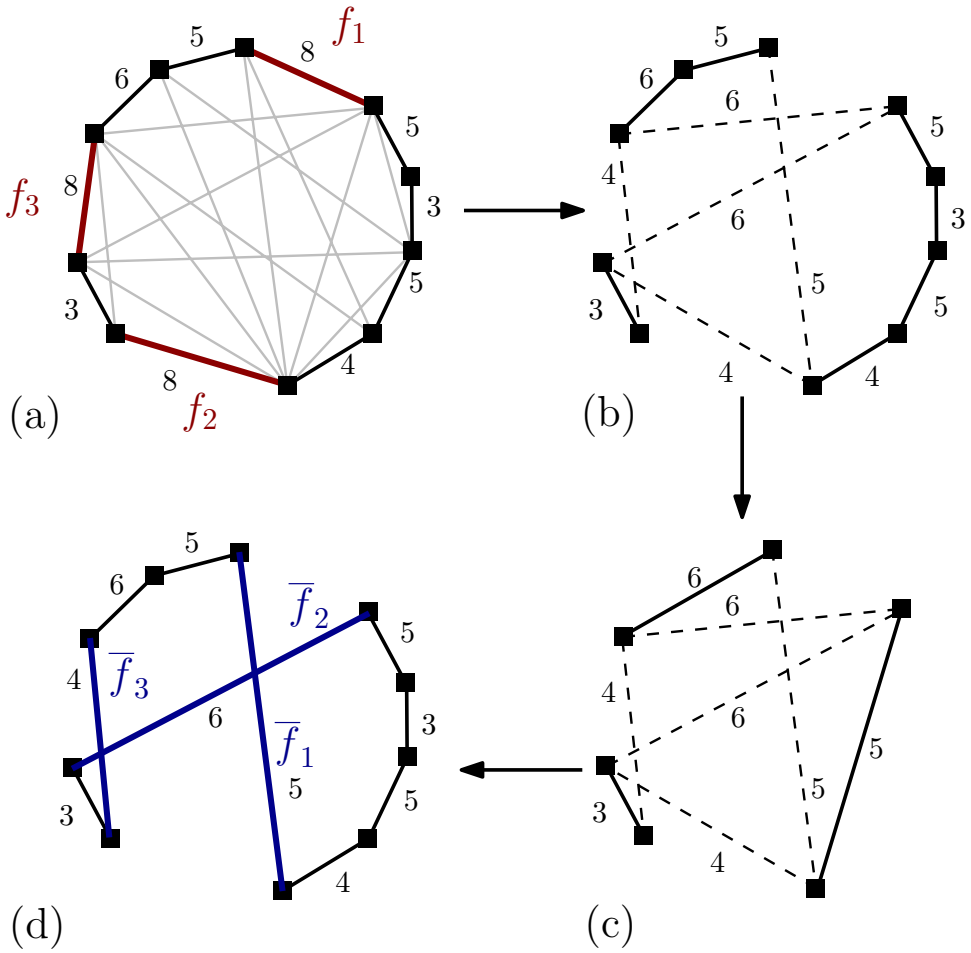


Figure 3.2. Illustration of a 3−opt move in 3−balanced. (1.a) represents a tour $\mathcal{H}$ whose largest and smallest edge costs are 8 and 3, respectively. We will remove all edges with max-cost 8 (f_1, f_2, f_3) from $\mathcal{H}$ and set $(l', u') = (l_{\mathcal{H}}, u_{\mathcal{H}} - 1) = (3, 7)$. (1.b) illustrates the remainder $\mathcal{H} \setminus F$ of the tour. The dash lines are the edges of $EC(F, l', u')$ where edges have two endpoints in $V(F)$, and costs belong to $[3, 7]$. (1.c) demonstrates a compressed version G' of G, in which paths in $\mathcal{H} \setminus F$ are considered as edges. The problem of reconnecting $\mathcal{H}$ in G is equivalent to the one in G'. (1.d) shows the resulting tour with a smaller max-min distance, i.e., 3.

We now present rules to select (F, l', u'). We create three variants of $k-$balanced corresponding to three selection rules for (F, l', u'): *$k-$balanced min*, *$k-$balanced max*, and *$k-$balanced extreme*.

Algorithm 9 Selection rule for $k-$balanced min/max

Input: A graph $G = (V, E)$, a tour $\mathcal{H}$, a constant k, and an extreme type ET.
Output: (F, l', u') where $F \subset \mathcal{H}$ and $[l', u'] \subsetneq [l_{\mathcal{H}}, u_{\mathcal{H}}]$.

1: **if** ET is min **then**
2: $\quad F \leftarrow \{e \in \mathcal{H} \mid c_e = l_{\mathcal{H}}\}, l' \leftarrow l_{\mathcal{H}} + 1, u' \leftarrow u_{\mathcal{H}}$
3: **else if** ET is max **then**
4: $\quad F \leftarrow \{e \in \mathcal{H} \mid c_e = u_{\mathcal{H}}\}, l' \leftarrow l_{\mathcal{H}}, u' \leftarrow u_{\mathcal{H}} - 1$
5: **end if**
6: **while** $|F| < k$ **do**
7: $\quad f \leftarrow \arg\max_{e=(i,j)\in\mathcal{H}} \mid \delta(\{i, j\}) \cap \delta(F) \cap \{e' \in E | l' \leq c_{e'} \leq u'\} \mid$
8: $\quad F \leftarrow F \cup \{f\}$
9: **end while**
10: **return** (F, l', u')

Algorithm 10 Selection rule for $k-$balanced extreme

Input: A graph $G = (V, E)$, a tour $\mathcal{H}$, and a constant k.
Output: (F, l', u') where $F \subset \mathcal{H}$ and $[l', u'] \subsetneq [l_{\mathcal{H}}, u_{\mathcal{H}}]$.

1: $F \leftarrow \emptyset$.
2: **while** $|F| < k$ **do**
3: $\quad removed\_cost \leftarrow \arg\min_{c_e \in C(\mathcal{H}\setminus F)} d(c_e, \mathcal{H})$
4: $\quad F \leftarrow F \cup \{e \in \mathcal{H} \mid c_e = removed\_cost\}$
5: **end while**
6: $l' \leftarrow \min_{e\in\mathcal{H}\setminus F} c_e$
7: $u' \leftarrow \max_{e\in\mathcal{H}\setminus F} c_e$
8: **return** (F, l', u')

Algorithm 9 describes the selection rule of (F, l', u') for $k-$balanced min/max. In these variants, we select F in such a way as to maximize the cardinality of $EC(F, l', u')$. We call this rule *the maximum candidate cardinality principle* (MCCP). In particular, for k-balanced min, we set $(l', u') = (l_{\mathcal{H}} + 1, u_{\mathcal{H}})$ and initialize F by all min-cost edges. At step i, an edge f_i in $\mathcal{H} \setminus F$ is added to the *current* F if it can increase the cardinality $EC(F, l', u')$ the most. More precisely, f_i is the edge that has the most incident edges having one end-vertex in $V(F)$ with costs between l' and u'. The selection procedure is repeated until the cardinality of F equals k. This selection rule is applied similarly for k-balanced max with two modifications: F initially is a set of all max-cost edges, and l', u' respectively equal $l_{\mathcal{H}}$ and $u_{\mathcal{H}} - 1$. Such a way to select (F, l', u') offers the uttermost cardinality of $EC(F, l', u')$ and thus increases the probability of $\overline{F}$'s existence. However, it slowly decreases the max-min distance at each iteration (the gain can be only 1 per k-exchange).

On the other hand, k-balanced extreme prioritizes dropping the max-min distance as fast as

possible. While k-balanced min/max chooses edges to remove, the removal rule of k-balanced extreme is cost-based. For an edge cost $C_i \in C(E)$, let $d(C_i, \mathcal{H}) := \min(|l_{\mathcal{H}} - C_i|, |u_{\mathcal{H}} - C_i|)$. The out-edge set F is iteratively constructed as follows. Initially, $F = \emptyset$. At each iteration, we select a cost $C_i \in C(\mathcal{H} \setminus F)$ that yields the smallest $d(C_i, \mathcal{H})$ and add all the edges with cost C_i in $\mathcal{H} \setminus F$ to F. The process terminates when the cardinality of F is at least k. The tuple (l', u') is set to $(\min_{e \in \mathcal{H} \setminus F} c_e, \max_{e \in \mathcal{H} \setminus F} c_e)$. This selection method can reduce the max-min distance substantially. However, it also decreases the cardinality of $EC(F, l', u')$ and thus decreases the possibility of finding the in-edge set $\overline{F}$. Algorithm 10 gives the formal description of the rule.

Table 3.2 summarizes the (F, l', u') selection rules of the three $k-$balanced variants.

	F	l'	u'
$k-$balanced min	min-cost edges and edges found by MCCP	$l_{\mathcal{H}} + 1$	$u_{\mathcal{H}}$
$k-$balanced max	max-cost edges and edges found by MCCP	$l_{\mathcal{H}}$	$u_{\mathcal{H}} - 1$
$k-$balanced extreme	extreme-cost edges and edges with smallest $d(c, \mathcal{H})$	$\min_{e \in \mathcal{H} \setminus F} c_e$	$\max_{e \in \mathcal{H} \setminus F} c_e$

Table 3.2. Selection rules of (F, l', u')

Notice that in all variants of $k-$balanced, we only consider one subset F to find $k-$exchange at each iteration. Although this setting can omit high-quality $k-$exchanges, it allows the algorithm to launch with many random initial tours and k's values within an acceptable amount of CPU time. Thus, we can still obtain reasonable, feasible solutions. To further improve the algorithm, when the number of min-cost edges or max-cost edges is at most 3, we search $3-$opt moves with all valid edge triples of the tour.

3.4.3 Edge elimination

In order to improve the efficiency of solving LP relaxations, it is helpful to eliminate edges that are not likely to be present in an optimal tour. It is important to note that the branch-and-cut algorithm aims to refine the incumbent solution. Consequently, if we can demonstrate that the occurrence of a specific edge would result in a worse tour compared to the current best tour, we can remove this edge to sparsify the graph and reduce the number of decision variables.

As proven in Lemma 3.4.2, if a tour contains an edge with cost C_i, its max-min distance is at least the length of the shortest biconnected interval compatible with C_i. Let $(\boldsymbol{x}_{\mathcal{H}^0}, l_{\mathcal{H}^0}, u_{\mathcal{H}^0})$ be the initial feasible solution corresponding to a tour $\mathcal{H}^0$ found by Algorithm 8. Then, edges with costs C_i satisfying $\gamma(C_i) > u_{\mathcal{H}^0} - l_{\mathcal{H}^0}$ can not be part of an optimal tour; otherwise, the max-min distance of this tour will be greater than $u_{\mathcal{H}^0} - l_{\mathcal{H}^0}$. By this observation, we can remove edges $e \in E$ such that $\gamma(c_e) > u_{\mathcal{H}^0} - l_{\mathcal{H}^0}$.

3.4.4 Variable fixing

To decrease the number of decision variables during the branch-and-cut algorithm, we fix some variables by adding corresponding cutting planes (e.g., add the inequality $x_e = 0$ if we want

to fix x_e to 0). Naturally, variables that can not help to improve the incumbent solution should be fixed to 0. In this section, we propose two heuristics to determine such variables: one based on biconnected intervals compatible with costs and one based on fixed costs at tree nodes. Throughout this section, let $(\overline{x}, \overline{u}, \overline{l})$ be the current incumbent solution of the enumeration tree.

3.4.4.1 Biconnected-interval-based variable fixing

Using the same arguments as in Section 3.4.3, edges with costs C_i such that $\gamma(C_i) \geq \overline{u} - \overline{l}$ can not be part of tours that are better than the current best tour in terms of the max-min distance. Thus, the variables corresponding to such edges can be permanently fixed to 0 in the remaining nodes of the tree. Therefore, when a new incumbent solution $(\overline{x}, \overline{u}, \overline{l})$ is obtained, we add the following inequalities to LP relaxations

$$x_e = 0 \quad \forall e \in E : \gamma(c_e) \geq \overline{u} - \overline{l}. \tag{3.4}$$

Obviously, the inequalities (3.4) are valid for the remainder of the enumeration tree.

3.4.4.2 Fixed-costs-based variable fixing

The second heuristic to fix variables is due to the fact that each node of the enumeration tree is associated with two disjoint edge sets F_0 and F_1 where F_0, F_1 consist of edges that have been fixed to 0 and 1, respectively. Given a node $\langle F_0, F_1 \rangle$, let $\mathcal{H}'$ be a tour found by this node or its descendants. If the max-min distance of $\mathcal{H}'$ is smaller than the current best objective value (i.e., $\overline{u} - \overline{l}$), then the edges of $\mathcal{H}'$ belong to either F_1 or $\{e \in E : S_{C(F1)} - (\overline{u} - \overline{l}) < c_e < I_{C(F_1)} + (\overline{u} - \overline{l})\}$ where $S_{C(F_1)}$ and $I_{C(F_1)}$ are respectively the maximum and minimum of $C(F_1)$. Thus, edges that are not in either of these two sets can be fixed to 0 in this node and its descendant. The inequalities corresponding to the fixing of these variables are

$$x_e = 0, \quad \forall e \in E : c_e \notin (S_{C(F_1)} - (\overline{u} - \overline{l}), I_{C(F_1)} + (\overline{u} - \overline{l})) \tag{3.5}$$

Since the validity of inequalities (3.5) depends on fixed costs at the node, these inequalities are only valid for the considered node and its descendants.

3.4.5 Separation algorithms and strategies

In branch-and-cut algorithms, cutting planes are generated at some nodes of the tree by solving separation problems. An efficient branch-and-cut algorithm relies on not only good separation algorithms but also deft separation strategies. In this section, we present separation procedures and strategies for generating subtour elimination constraints and local bounding cuts. We first denote by (x^i, u^i, l^i) a fractional solution of an LP relaxation associated with a tree node.

3.4.5.1 Subtour elimination constraints

Recall that subtour elimination inequalities have the form $\sum_{e \in \delta(S)} x_e \geq 2$ where $S \subset V$. To find subtour elimination constraints violated by x^i, one can construct a graph $G^i = (V, E^i)$ with edge set $E^i = \{e \in E \mid x^i_e > 0\}$. A weight associated with $e \in E^i$ is x^i_e. By this setting, a violated

subtour elimination constraint is a cut whose weight is less than 2 in G^i. Such a cut can be found via a Gomory-Hu tree [51] of G^i, built from $|V| - 1$ max-flow computation.

Since solving subtour's separation problem is computationally expensive and can provide no cutting planes, we only generate these inequalities at every 100 nodes instead of every node in the enumeration tree.

3.4.5.2 Local bounding cuts

At a node associated with ordered edge set pair $< F_0, F_1 >$, one can generate at most $O(|E|)$ local bounding cuts. However, if we generate all possible local bounding cuts at every node, the subproblems will be enormous and hard to solve. Thus, we only generate local bounding cuts with variables x_e such that $0 < x_e^i < 1$ and $I_{C(F_1)} < M_e$. In addition, since the local bounding cuts are mainly for the optimality phase, we only generate them when the IP relative gap is less than 0.5 at every 10 nodes.

3.5 Computational results

In this section, we present experimental results to assess the efficiency of our branch-and-cut algorithm. All the experiments are conducted on a PC Intel Core i7-10700 CPU 2.9GHz and 64 GB RAM. The algorithm is implemented in Python using CPLEX 12.10 with default setting and one solver thread. The CPU time limit for exploring the enumeration tree is set to 10800 seconds. To make comparisons with the DT algorithms [13], we use the identical testbed of 65 TSPLIB instances [46]. These instances include graphs with 14 to 493 vertices.

To verify the graph's biconnectivity in the lower bounding algorithm (i.e., Algorithm 7), we use a depth-first-search algorithm implemented by the Networkx algorithm [52].

We initialize tours for $k-$balanced algorithms by permutations of $\{1, \ldots, |V|\}$ since the testing instances are complete graphs. The problem of completing a Hamiltonian cycle to find $k-$exchanges is resolved by MILP through using CPLEX 12.10. To initialize good upper bounds for the branch-and-cut algorithm, we execute $k-$balanced extreme and $3-$balanced with 10 random tours. For instances with fewer than 50 vertices, only $3-$balanced is performed. The k values are in the set $\{10, 20, \ldots, \mathcal{K}\}$ where $\mathcal{K}$ is defined based on instance sizes in Table 3.3. During the branch-and-cut algorithm, we execute $k-$balanced min/max with $k = \mathcal{K}$ and $3-$balanced to improve the incumbent solutions.

Graph size ($\|V\|$)	$\|V\| < 50$	$50 \leq \|V\| < 100$	$100 \leq \|V\| < 200$	$\|V\| \geq 200$
$\mathcal{K}$	–	30	50	100

Table 3.3. The values of $\mathcal{K}$ correspond to instance sizes.

We first select 12 instances from the test set to analyze the impact of our additional components in the branch-and-cut algorithm. The initial set comprises four small-sized instances (gr21, hk48, eil75, gr96), four medium-sized instances (pr136, si175, d198, tsp225) and four large-sized instances(a280, lin318, pcb442, d493). The first experiment in Section 3.5.1 aims

Instance	Our B&C				CPLEX			
	Obj	Gap(%)	CPU(s)	Nodes	Obj	Gap(%)	CPU(s)	Nodes
gr21	115	0.0	0.6	0	115	0.0	0.2	298
hk48	156	0.0	4.3	157	156	0.0	5.7	3389
eil76	2	0.0	6.2	390	2	0.0	5241.6	470000
gr96	314	0.0	93.9	1130	314	0.0	143.1	12957
pr136	126	0.0	62.5	1126	126	0.0	243.8	15709
si175	7	**0.0**	150.6	3854	7	28.6	10800.0*	113245
d198	1122	**0.0**	2424.5	16892	1122	58.7	10801.4*	62677
tsp225	6	0.0	135.0	682	6	0.0	3955	28550
a280	3	0.0	196.8	481	3	0.0	5319.6	31856
lin318	31	**0.0**	499.3	1591	641	100.0	10804.8*	43700
pcb442	27	**0.0**	9013.8	1592	283	100.0	10805.6*	22712
d493	1193	**0.0**	4114.4	7399	1628	100.0	10808.1*	8935
Average		**0.0**	**1391.8**	**2941.2**		32.3	5744.1	67835.7

Table 3.4. Comparison between the two algorithms on the 12 TSPLIB instances

at comparing our branch-and-cut algorithm to the commercial solver CPLEX 12.10, a general-purpose branch-and-cut algorithm. Then, Section 3.5.2 evaluates the impact of the components: local bounding cuts, the lower bounding algorithm, and the $k-$balanced algorithm. Finally, in Section 3.5.3, the entire testbed's results are shown with a comparison to the results of the DT algorithms reported in [13].

3.5.1 The effectiveness of the proposed branch-and-cut algorithm

In the first experiment, we compare our algorithm with the commercial solver CPLEX in solving (MILP-BTSP).

Table 3.4 reports the results of the two algorithms on the initial test set. Column *"Instance"* displays the instance's names where the number at the end stands for the number of graph vertices. The results of each algorithm contain the best objective value found by the algorithm within the CPU time limit (subcolumn *"Obj"*), the current IP relative gap (subcolumn *"Gap(%)"*), the running time in seconds (subcolumn *"CPU(s)"*), and the number of nodes in the enumeration tree (subcolumn *"Nodes"*). Notice that the running time includes the time spent on the initialization step and the enumeration tree exploration. Instances whose running times are marked with an asterisk (*) are instances that cannot be solved to proven optimality within the CPU time limit.

Numerical results illustrate that our branch-and-cut algorithm outperforms CPLEX. Indeed, within the CPU time limit, our algorithm can solve all 12 instances to proven optimality, whereas CPLEX can only solve 7 out of 12 instances to proven optimality. More precisely, CPLEX fails to prove the solution optimality for si175 and d198, and could not find an optimal solution for lin318, pcb442, and d493. Among the 12 instances, there is only one instance (gr21) on which our algorithm performs slower; for the rest, our algorithm solves the problems 4 times faster on average than CPLEX. Furthermore, our algorithm's average tree size is 23 times smaller than that of CPLEX.

Instance	Full		Full Local cuts		Full Lower bounding		Full $k-$balanced	
	Gap(%)	CPU(s)	Gap(%)	CPU(s)	Gap(%)	CPU(s)	Gap(%)	CPU(s)
gr21	0.0	0.6	0.0	0.5	0.0	0.6	0.0	0.6
hk48	0.0	4.3	0.0	3.8	0.0	3.8	0.0	16.5
eil76	0.0	6.2	0.0	471.3	0.0	1427.2	0.0	251.0
gr96	0.0	93.9	0.0	57.1	0.0	110.2	0.0	151.1
pr136	0.0	62.5	0.0	94.0	0.0	78.6	0.0	161.8
si175	0.0	150.6	0.0	3169.5	71.4	10806.9*	0.0	3579.2
d198	0.0	2424.5	0.0	1537.8	44.9	10810.3*	27.6	10824.9*
tsp225	0.0	135.0	0.0	991.6	0.0	3096.0	0.0	4232.0
a280	0.0	196.8	25.0	10826.9*	100.0	10825.6*	0.0	10074.2
lin318	0.0	499.3	0.0	461.8	0.0	1014.8	96.1	10835.7*
pcb442	0.0	9013.8	23.5	10899.8*	100.0	10858.9*	98.0	10847.4*
d493	0.0	4114.4	0.0	9118.0	0.0	6568.6	98.2	10862.8*
Average	**0.0**	**1391.8**	4.0	3136.0	26.4	4633.5	26.7	5153.1

Table 3.5. Computational results of the algorithm variants

3.5.2 Impact of local cuts, lower bounding, and $k-$balanced components

The aim of this section is to evaluate the effectiveness of three additional components: local bounding cuts, the lower bounding algorithm, and the $k-$balanced algorithm. Toward this end, we create four algorithm variants. The *Full* variant is the complete version that uses all three components. The *Full x* variant represents a version excluding the component x, such as *Full Local cuts* omits local bounding cuts.

The computational results in Table 3.5 indicate that all proposed components are crucial for the efficiency of the branch-and-cut algorithm. Excluding any of these components can dramatically increase the algorithm's running time and lead to failure in solving several instances to proven optimality. We can order the impact of the components as follows: $k-$balanced $>$ Lower bounding $>$ Local cuts. More precisely, when the $k-$balanced algorithm is omitted, the computing time increases the most, i.e., 3.2 times compared to the Full version. Then, the absence of the lower bounding algorithm results in a slowdown of 2.7 times, and the lack of local bounding cuts leads to a 1.6 times decrease in speed.

Instance	DT algorithms [13]			Our B&C			
	LB	Obj	CPU(s)	Initial LB	Initial UB	Obj	CPU(s)
burma14	120	134	0.02	120	134	**134**	0.3
ulysses16	837	868	0.05	173	868	**868**	0.5
gr17	94	119	0.07	80	129	**119**	0.5
gr21	110	115	0.07	65	120	**115**	0.6
ulysses22	837	868	0.29	157	868	**868**	0.6
gr24	33	33	0.1	33	45	33	0.7
fri26	21	21	0.04	21	25	21	0.5
bayg29	23	29	0.3	23	34	**29**	0.8
bays29	36	38	0.37	36	49	**38**	1.9
dantzig42	13	13	0.66	13	21	13	1.7
swiss42	14	14	0.22	14	32	14	1.7
att48	156	192	5.1	133	223	**190**$^{\downarrow}$	3.9
gr48	46	46	2.26	46	96	46	2.9
hk48	138	156	5.06	133	189	**156**	4.3
eil51	3	3	0.35	3	6	3	1.5
berlin52	139	149	5.33	113	151	**149**	5.5
brazil58	912	1124	12.51	912	1124	**1097**$^{\downarrow}$	7.7
st70	5	5	1.5	5	6	5	1.9
eil76	2	2	0.63	2	5	2	6.2
pr76	498	522	7.03	498	1015	**522**	8.6
gr96	281	314	83.51	281	561	**314**	93.9
rat99	5	5	2.84	5	9	5	9.2
kroA100	137	137	46.39	137	463	137	83
kroB100	129	145	47.96	129	471	**145**	65.7
kroC100	120	133	51.08	120	509	**133**	72.7
kroD100	140	140	42.54	137	269	140	422
kroE100	137	139	48.61	137	452	**139**	60.9
rd100	43	43	18.68	43	53	43	10.3
eil101	2	2	1.73	2	3	2	3.5
lin105	95	100	89.26	95	183	**100**	26.9
pr107	877	900	29.58	53	3645	**877**$^{\downarrow}$	25.2
gr120	27	31	20.39	27	94	**31**	67.9
pr124	364	408	258.94	364	731	**406**$^{\downarrow}$	93.2
bier127	2915	3023	186.82	874	3459	**2925**$^{\downarrow}$	29.8
ch130	18	22	25.91	17	60	**22**	56.7
pr136	103	126	29.95	103	1149	**126**	62.5
gr137	403	424	352.98	354	825	**424**	256.3
pr144	259	259	126	259	449	259	43
ch150	17	17	23.61	17	33	17	196.9
kroA150	89	91	120.43	89	452	**91**	122.2
kroB150	103	109	150.36	100	454	**109**	83.7
pr152	59	59	155.6	59	378	59	63.3
u159	142	142	38.08	135	822	142	1933.8
si175	7	7	50.76	5	21	7	150.6
brg180	0	0	0.31	0	0	0	2.8
rat195	4	4	8.58	4	16	4	499.6
d198	1105	1125	240.34	830	1355	**1122**$^{\downarrow}$	2424.5
kroA200	71	76	247.38	71	599	**76**	1607.7
kroB200	81	82	292.5	81	522	**82**	1242.7

gr202	778	875	1039.92	69	933	**787**↓	289.2
ts225	21	21	19.63	0	696	21	503.1
tsp225	6	6	50.05	6	21	6	135
pr226	450	504	1260.9	450	704	**504**	123.5
gr229	675	742	1737.24	622	1660	**706**↓	849.3
gil262	3	3	46.51	3	7	3	99.5
pr264	238	415	1226.01	238	3255	**340**↓	7386.6
a280	3	3	24.56	3	16	3	196.8
pr299	89	89	473.29	89	363	89	4258.6
lin318	31	31	430.65	31	133	31	499.3
rd400	11	11	269.08	11	17	11	243.3
fl417	199	260	4047.06	82	359	229↓	10931.2*
gr431	1943	2195	13038.11*	502	2876	**1962**↓	6555.5
pr439	810	1548	11967.86*	256	2583	994↓	11254.4*
pcb442	26	27	571.16	26	161	**27**	9013.8
d493	1191	1423	3136.15	34	1592	**1193**↓	4114.4

Table 3.6. Numerical results of the DT and branch-and-cut algorithms on 65 TSPLIB instances. Instances with the bold objective value are solved to proven optimality for the first time, and instances with objective values marked by ↓ are ones that our algorithm can provide better solutions. For the instances fl417 and pr439, which can not be solved to proven optimality within the CPU time limit, their current IP relative gap are 64.2% and 74.3%, respectively.

3.5.3 Comparison to the double-threshold-based algorithms

Finally, we present the results of the branch-and-cut algorithm on the entire testbed with a comparison to the double-threshold (DT) algorithms [13]. To ensure fairness in the comparisons, we reproduce the lower bounding algorithm and DT algorithms using the code furnished by the original authors[1]. Note that we only implement the modified double-threshold (MDT) algorithm and iterative bottleneck (IB) algorithm since these algorithms are reported to be the best among the four heuristic variants proposed in [13]. The CPU time limit for the DT algorithms is also set to 10800 seconds.

Table 3.6 represents the algorithm's results. Column *"DT algorithms"* corresponds to the DT algorithms' results. Subcolumn *"LB"* reports lower bounds on the optimal value of the BTSP obtained by the procedure proposed in [13]. Subcolumn *"Obj"* represents the best objective values found by the MDT or IB algorithm. Subcolumn *"CPU(s)"* gives the total time in seconds spent by the lower bounding procedure and the DT algorithms. In column *"Our B&C"* which represents the results of the proposed branch-and-cut algorithm, subcolumns *"Initial LB"* and *"Initial UB"* respectively indicate lower and upper bounds obtained by our lower bounding and $k-$balanced algorithms in the initialization step.

As shown in Table 3.6, the DT algorithms can find an optimal solution for 52 over 65 instances, but only 27 solutions are certified for their optimality. In contrast, our branch-and-cut algorithm can solve to proven optimality 63 out of 65 instances within the CPU time limit, and thus provide optimality certificates of 36 solutions for the first time. Furthermore, for 13 of the 65 problems - mainly large-sized instances, our algorithm obtains solutions better than the DT

[1]See at https://github.com/johnlarusic/arrow

algorithms. Although the two instances fl417 and pr439 can not be solved optimally within the time limit, their best objective values so far are significantly smaller than the DT algorithms' ones.

3.6 Conclusion

In this chapter, we proposed a branch-and-cut algorithm for balanced combinatorial optimization. To deal with the main challenge of estimating the largest and smallest components of the solution, we introduced local bounding cuts, a new class of local cuts that can be applied to any balanced combinatorial optimization problem. We then specialized in the branch-and-cut algorithm for the BTSP, an NP-hard problem. We developed additional mechanisms to locate the highest and smallest edge costs (i.e., edge elimination and variable fixing techniques) and algorithms to initialize lower and upper bounds on the optimal value of the BTSP. To evaluate the effectiveness of our algorithm, we conducted experiments on TSPLIB instances with less than 500 vertices. For 63 out of 65 instances, we obtained optimal solutions. For 13 of the 65 instances - mainly large-sized ones, our algorithm provided solutions with smaller objective values compared with the previous work in the literature [13]. For solving exactly large-scale instances of thousands of vertices, more mechanisms of tightening lower and upper bounds would be needed. Interesting directions for future research would be the investigation of new classes of local cuts for balanced combinatorial optimization.

Chapter 4

Algorithmic aspects of fair combinatorial optimization by Ordered Weighted Average

Summary

In the preceding chapter, the issue of fairness is addressed by minimizing the disparity in value between the most expensive and least expensive components chosen. Clearly, this objective solely depends on the extreme costs; thus, vectors with the same max-min distance can not be discriminated. However, in practical terms, the vector $(100, 1, ..., 1, 1)$ is more desirable than $(100, 100, ..., 100, 1)$. To resolve this, one can employ an aggregation function that assesses vectors in a manner favoring lower values.

Note that in fair combinatorial optimization, our primary focus is on the distribution of outcome values, disregarding their orders. In other words, we are more concerned with the set of solution component values than with which components take a particular value. An approach considering this observation is OWA combinatorial optimization, where the values of components are aggregated by the OWA operator. This operator distinguishes itself from the traditional weighted average in that coefficients are associated with ordered positions rather than specific attributes.

The OWA operator is commonly used in multi-objective optimization problems where the criteria are measured on the same scale and can be directly compared. Such problems include network dimensioning [53], broadcasting network problems [54], and multi-objective Markov decision processes [55], to name a few. In the context of fair combinatorial optimization, each component used in the solution is viewed as an objective. For example, in an optimal solution of the TSP, the cost of each selected edge is an individual objective. Naturally, the vector of component values should be both Pareto-optimal (meaning it cannot be improved on all components simultaneously) and fair.

In order to obtain such vectors, the OWA weights are selected to be strictly decreasing and positive. In this case, the OWA is referred to as the Generalized Gini Index (GGI) [56], a well-known inequality measure in economics. The GGI fulfills natural properties to encode fairness, including the Pigou-Dalton principle of transfers, which states that transferring any small amount from one outcome to a comparatively worse-off outcome will result in a more favored outcome vector. Moreover, the GGI increases with respect to Pareto-dominance, ensuring that every optimal solution to the GGI is also a Pareto-optimal solution.

Due to the ordering operator in the objective function, OWA combinatorial optimization problems are non-linear, even if their original constraints are linear. Several MILP formulations have been proposed in the literature to linearize the OWA. For the GGI with decreasing weights, there exist two ways to linearize it: one proposed by Ogryczak et al. [1] using the cumulative ordered achievement vector (formulation O-MILP) and the other given by Chassein et al. [2] employing the permutahedron (formulation C-MILP). Although C-MILP was reported to be more efficient than O-MILP for some continuous optimization problems [2], it remains uncertain which formulation is best for fair combinatorial optimization, as integrating linearization methods into the formulations of OWA combinatorial optimization may imply additional difficulties. For instance, Lesca et al. [57] proved that the OWA assignment problem is NP-hard.

In this chapter, we provide a comparative study of the two MILP formulations for OWA combinatorial optimization. Our study contains theoretical analysis and experimental results on a specific combinatorial optimization problem (i.e., the TSP). In particular, we prove that O-MILP and C-MILP are equivalent in terms of linear relaxations and estimate the GGI value

quality of the optimal solution of classical combinatorial optimization. We also experimentally compare O-MILP, C-MILP, and primal-dual heuristics [8] for solving the OWA TSP. Interestingly, although O-MILP uses more variables than C-MILP, its performance on the OWA TSP is better than that of C-MILP.

The structure of this chapter is as follows. We first recall the definition of OWA combinatorial optimization in Section 4.1, followed by two linearization methods in Section 4.2. In Section 4.3, we provide a theoretical analysis of the relation between the MILP formulations and a quality evaluation of the optimal solution of the classical combinatorial optimization. Section 4.4 presents a generic method based on the primal-dual algorithm in [8] as a faster alternative to deal with large-sized instances. Experimental results to evaluate the methods are shown in Section 4.5. Finally, our conclusions are discussed in Section 4.6.

4.1 OWA combinatorial optimization

We consider a combinatorial optimization problem whose feasible set $\mathcal{X} \in \{0,1\}^m$ can be described as:

$$\begin{aligned} \boldsymbol{A}\boldsymbol{x} &\leq \boldsymbol{b} \\ \boldsymbol{x} &\in \{0,1\}^m \end{aligned}$$

where $\boldsymbol{A} \in \mathbb{R}^{K\times m}$, $\boldsymbol{b} \in \mathbb{R}^K$, K, m are positive integers, and $\boldsymbol{x}$ is the vector of decision variables. We refer to the values of components selected in a feasible solution $\boldsymbol{x} \in \mathcal{X}$ (i.e., components corresponding to $x_i = 1, i \in [m]$) as a value vector $\boldsymbol{v} = (v_1, \ldots, v_n)$. We restrict the number of selected components to being constant for all feasible solutions. Let $\boldsymbol{C}$ be an $n \times m$ matrix that maps the feasible set $\mathcal{X}$ into the value set $\mathcal{V} \subseteq \mathbb{R}^n$, namely that $\boldsymbol{v} = \boldsymbol{C}\boldsymbol{x}$.

Combinatorial optimization problems typically seek a feasible solution $\boldsymbol{x}$ that optimizes an aggregation function of value vectors. For example, in the TSP, value vector $\boldsymbol{v}$ represents the costs of edges in the tour, and the aggregation function is the sum of the elements in $\boldsymbol{v}$. Formally, a combinatorial optimization problem can be formulated as follows:

$$\begin{aligned} &\min\ f(\boldsymbol{v}) && \text{(4.1a)}\\ &\text{s.t. } \boldsymbol{v} = \boldsymbol{C}\boldsymbol{x} && \text{(4.1b)}\\ &\quad \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} && \text{(4.1c)}\\ &\quad \boldsymbol{x} \in \{0,1\}^m && \text{(4.1d)} \end{aligned}$$

where f is an aggregation function. In classical optimization problems, the objective is typically minimizing the total values of $\boldsymbol{v}$'s components. In that case, the aggregation function is the sum, and the problem can be rewritten as:

$$\begin{aligned} &\min \sum_{i\in[n]} v_i && \text{(4.2a)}\\ (\text{Min}-\mathcal{P}) \quad &\text{s.t } \boldsymbol{v} = \boldsymbol{C}\boldsymbol{x} && \text{(4.2b)}\\ &\quad \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} && \text{(4.2c)}\\ &\quad \boldsymbol{x} \in \{0,1\}^m. && \text{(4.2d)} \end{aligned}$$

When dealing with the issue of fairness, we focus on the distribution of values in the vector $\boldsymbol{v}$. Thus, solutions with the same set of values, though potentially in a different order, should be treated as identical. In order to ensure this, the aggregation function must satisfy:

$$f(v_1, \ldots, v_n) = f(v_{\tau(1)}, \ldots, v_{\tau(n)}) \tag{4.3}$$

for any permutation τ of $[n]$. One of such aggregation function is the OWA operator, introduced by Yager [20]. In the OWA operator, the weights are associated with the ordered attribute values (i.e., the largest value is multiplied by the first weight, the second largest value by the second weight, and so on). Hence, it is permutation-independent. Formally, this operator can be defined as follows.

Definition 4.1.1. [20] Given a vector $\boldsymbol{v} \in \mathbb{R}^n$, the OWA value of $\boldsymbol{v}$ is defined by:

$$\mathrm{OWA}_{\boldsymbol{w}}(\boldsymbol{v}) = \sum_{k \in [n]} w_k \theta_k(\boldsymbol{v}) \tag{4.4}$$

where $\boldsymbol{w} = (w_1, \ldots w_n) \in [0,1]^n$ and $\theta_k(\boldsymbol{v})$ is the kth largest component of $\boldsymbol{v}$.

Note that Definition 4.1.1 presented here differs from the original definition introduced by Yager [20], as the weights are not necessarily normalized. This modification simplifies the transformation used later on without loss of generality.

The OWA allows for modeling various aggregation functions, such as the minimum ($w_k = 0, \forall k \in [n-1]$ and $w_n = 1$), average ($w_k = 1/n, \forall k \in [n]$), maximum ($w_1 = 1$ and $w_k = 0, \forall k \neq n$), and sum ($w_k = 1, \forall k \in [n]$). In cases where the weights $\boldsymbol{w}$ are strictly decreasing and positive, the OWA is referred to as the *Generalized Gini Index* (GGI) [56], a well-studied measure to represent the income or wealth inequality in economics. We denote the GGI with weights $\boldsymbol{w}$ by $G_{\boldsymbol{w}}$.

Thanks to the weight restrictions, the GGI increases with respect to Pareto dominance, i.e., if $\boldsymbol{v} \in \mathbb{R}^n$ Pareto-dominates $\boldsymbol{v}' \in \mathbb{R}^n$ ($v_i \geq v'_i \, \forall i \in [n], \exists j \in [n], v_j > v'_j$) then $G_{\boldsymbol{w}}(\boldsymbol{v}) > G_{\boldsymbol{w}}(\boldsymbol{v}')$. Due to this property, every optimal solution to (4.1) is a Pareto-optimal (*efficient*) solution. This is consistent with the minimization of all individual solution components.

More importantly, the GGI can encode the Pigou-Dalton principle of transfers [58, 59], an important property when measuring inequality. Intuitively, the Pigou-Dalton principle states that transferring an amount from a higher value component to a lower one without reversing their relative orders obtains a fairer value vector. The GGI is designed to decrease with such transfers, reflecting the reduced inequality. Formally, for any $\boldsymbol{v} \in \mathbb{R}^n$ where $v_i < v_j$ and $\epsilon \in (0, v_j - v_i)$, then $G_{\boldsymbol{w}}(v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n) > G_{\boldsymbol{w}}(v_1, \ldots, v_i + \epsilon, \ldots, v_j - \epsilon, \ldots, v_n)$. This principle is equivalent to requiring $G_{\boldsymbol{w}}$ is strictly Schur-concave [60].

The equitability property encoded by the GGI can be illustrated through the Lorenz curve [61], a widely used graphical representation of the distribution of income within a population, as in Figure 4.1. Let $\bar{\theta}_k(\boldsymbol{v}) = \sum_{i \in [k]}$ be the kth Lorenz component of $\boldsymbol{v}$. Then, the GGI can be rewritten as the weighted sum of Lorenz components, i.e.,

$$G_{\boldsymbol{w}}(\boldsymbol{v}) = \sum_{k \in [n]} w'_k \bar{\theta}_k(\boldsymbol{v}) \tag{4.5}$$

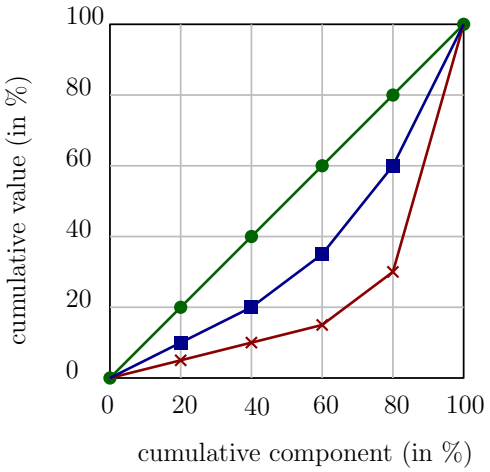


Figure 4.1. Lorenz curves. The green line represents perfect equality. The red and blue lines depict the Lorenz curves for vector $\boldsymbol{v}$ before and after making a Pigou-Dalton transfer, respectively.

where $w'_k = w_k - w_{k+1}$ for $k \in [n-1]$ and $w'_n = w_n$. A perfectly equal distribution of value vectors would be one in which every component has the same value. In that case, the highest $N\%$ of components should have $N\%$ of the total values. This can be depicted by the straight line $y = x$, referred to as the *line of perfect equality*. Therefore, the closer the Lorenz curve of vector $\boldsymbol{v}$ is to this line, the more equally distributed the $\boldsymbol{v}$'s values are. When taking a Pigou-Dalton transfer, the resulting vector's Lorenz curve moves closer to the perfect equality line. This represents a more equal distribution of the vector's values through the transfer.

In summary, the GGI can encode both the notions of efficiency (with respect to Pareto dominance) and fairness (with respect to the Pigou-Dalton transfer). Thus, in this chapter, we investigate OWA combinatorial optimization, which uses the GGI aggregation in the objective function, i.e.,

$$
(\text{OWA}-\mathcal{P}) \quad
\begin{aligned}
&\min\ G_w(\boldsymbol{v}) && \text{(4.6a)}\\
&\text{s.t}\ \boldsymbol{v} = \boldsymbol{C}\boldsymbol{x} && \text{(4.6b)}\\
&\quad \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} && \text{(4.6c)}\\
&\quad \boldsymbol{x} \in \{0,1\}^m && \text{(4.6d)}
\end{aligned}
$$

For illustration, we now present the concept of OWA combinatorial optimization in the context of TSP.

Example 4.1.2. Given a graph $G = (V, E)$ where $V := [n] = \{1, 2, \ldots, n\}$, $|E| = m$, and a cost vector $\boldsymbol{c} \in \mathbb{R}^m$ associated with the edge set E, the TSP seeks the shortest tour which visit each city exactly once and returns to the origin city. Therefore, $\boldsymbol{v}$ contains the costs of n edges selected in the tour.

To represent $\boldsymbol{v}$, we consider a directed version $G^d = (V, E^d)$ of G. In particular, each edge $e = (i, j) \in E$ becomes two arcs (i, j) and (j, i) with cost c_e, i.e., $c_{ij} = c_{ji} = c_e$. For each arc (i, j), let x_{ij} be a binary variable to represent the occurrence of (i, j) in a directed tour of G^d. Denote $N^-(i) = \{j \in [n] | (j, i) \in E^d\}$ and $N^+(i) = \{j \in [n] | (i, j) \in E^d\}$ the in-neighbor and out-neighbor sets of vertex $i \in [n]$. By the fact that each vertex i has only

one incoming arc in the directed tour, i.e., $\sum_{j\in N^-(i)} x_{ji} = 1, \forall i \in [n]$, $\boldsymbol{v}$ can be represented by $(\sum_{j\in N^-(1)} c_{j1}x_{j1}, \ldots, \sum_{j\in N^-(n)} c_{jn}x_{jn})$.

Instead of minimizing the sum objective function, OWA TSP optimizes the GGI objective function $\sum_{i\in[n]} w_i\theta_i(\boldsymbol{v})$, which can encode both the efficiency and fairness of $\boldsymbol{v}$. A MILP formulation for OWA TSP based on the subtour polytope [47] can be written as follows:

$$
\begin{aligned}
&\min\ G_{\boldsymbol{w}}(\boldsymbol{v}) && && (4.7a)\\
&\text{s.t } v_i = \sum_{j\in N^-(i)} c_{ji}x_{ji} && \forall i \in [n] && (4.7b)\\
&\sum_{j\in N^-(i)} x_{ji} = 1 && \forall i \in [n] && (4.7c)\\
&\sum_{j\in N^+(i)} x_{ij} = 1 && \forall i \in [n] && (4.7d)\\
&\sum_{i,j\in Q,(i,j)\in E^d} x_{ij} \le |Q| - 1 && \forall Q \subset V && (4.7e)\\
&\boldsymbol{x} \in \{0,1\}^{2m}. && && (4.7f)
\end{aligned}
\qquad \text{(OWA}-\text{TSP)}
$$

The OWA TSP is NP-hard since it is reduced from finding a Hamiltonian cycle problem.

Because of the objective function, OWA combinatorial optimization is non-linear. More precisely, the ordering operator θ in the OWA function causes OWA combinatorial optimization to become non-linear even if the original constraints are linear. Thankfully, the OWA function can be linearized by two methods studied in [1] and [2], which are presented now.

4.2 MILP formulations

4.2.1 Formulation O-MILP [1]

In [1], Ogryczak et al. proposed a method to linearize the OWA function using the Lorenz components of $\theta(\boldsymbol{v})$. Notice that when fixing $\boldsymbol{v}$, $\overline{\theta}_k(\boldsymbol{v})$ can be computed as a solution to a knapsack problem:

$$
\overline{\theta}_k(\boldsymbol{v}) = \max\left\{\sum_{i\in[n]} v_i a_{ki} \,|\, a_k \in [0,1]^n, \sum_{i\in[n]} a_{ki} = k\right\} \tag{4.8}
$$

To integrate into a LP where $\boldsymbol{v}$ is also variable, we take a dual of (4.8), which is

$$
\begin{aligned}
\overline{\theta}_k(\boldsymbol{v}) = \min\quad & kr_k + \sum_{i=1}^{n} d_{ki} && \\
\text{s.t}\quad & r_k + d_{ki} \ge v_i && \forall i \in [n]\\
& d_{ki} \ge 0 && \forall i \in [n].
\end{aligned}
\tag{4.9}
$$

Combining with (4.5), we get a MILP formulation for (OWA$-\mathcal{P}$), called O-MILP:

$$
\begin{aligned}
& \min \sum_{k\in[n]} w'_k(kr_k + \sum_{i\in[n]} d_{ki}) && && (4.10a)\\
& \text{s.t } r_k + d_{ki} \geq v_i && \forall k,i \in [n] && (4.10b)\\
(\text{O-MILP}) \quad & d_{ki} \geq 0 && \forall k,i \in [n] && (4.10c)\\
& \boldsymbol{v} = \boldsymbol{C}\boldsymbol{x} && && (4.10d)\\
& \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} && && (4.10e)\\
& \boldsymbol{x} \in \{0,1\}^m && && (4.10f)
\end{aligned}
$$

Before presenting the second formulation, we exploit O-MILP's structure to get useful information for later analysis. Toward this end, we consider a dual of O-MILP's continuous relaxation (denoted as $(\mathcal{D}_O)$) which has the following form:

$$
\begin{aligned}
& \max \; -\boldsymbol{u}^T\boldsymbol{b} - \sum_{k\in[m]} t_k && && (4.11a)\\
& \text{s.t. } \sum_{i\in[n]} y_{ki} = kw'_k && \forall k \in [n] && (4.11b)\\
& y_{ki} \leq w'_k && \forall k,i \in [n] && (4.11c)\\
(\mathcal{D}_O) \quad & \sum_{k\in[n]} y_{ki} + z_i = 0 && \forall k \in [n] && (4.11d)\\
& (\boldsymbol{u}\boldsymbol{A})_k - (\boldsymbol{z}\boldsymbol{C})_k + t_k \geq 0 && \forall k \in [m] && (4.11e)\\
& y_{ki} \geq 0 && \forall k,i \in [n] && (4.11f)\\
& t_k \geq 0 && \forall k \in [m] && (4.11g)\\
& u_k \geq 0 && \forall k \in [K]. && (4.11h)
\end{aligned}
$$

Interestingly, if we fix $\boldsymbol{y}$ and take a dual of $(\mathcal{D}_O)$, we obtain a continuous relaxation of (Min$-\mathcal{P}$) with modified costs, i.e.

$$
\begin{aligned}
& \min \sum_{i\in[n]} (\sum_{k\in[n]} y_{ik})v_i && && (4.12a)\\
(RP_{\boldsymbol{y}}) \quad & \text{s.t. } \boldsymbol{v} = \boldsymbol{C}\boldsymbol{x} && && (4.12b)\\
& \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} && && (4.12c)\\
& \boldsymbol{x} \in [0,1]^m && && (4.12d)
\end{aligned}
$$

As a consequence, a solution $(\boldsymbol{x},\boldsymbol{v})$ for the formulation $(RP_{\boldsymbol{y}})$ with integrality conditions $\boldsymbol{x} \in \{0,1\}^m$ is also feasible for O-MILP (in the sense that one can also find $(\boldsymbol{r},\boldsymbol{d})$ such that $(\boldsymbol{r},\boldsymbol{d},\boldsymbol{x},\boldsymbol{v})$ is feasible for O-MILP and its corresponding objective function is equal to $G_{\boldsymbol{w}}(\boldsymbol{v})$). We denote $(P_{\boldsymbol{y}})$ the discrete version of $(RP_{\boldsymbol{y}})$ where $\boldsymbol{x} \in \{0,1\}^m$.

4.2.2 Formulation C-MILP [2]

An alternative approach to linearize $G_{\boldsymbol{w}}(\boldsymbol{v})$, studied in [2], starts from an observation that if we permute $\boldsymbol{w}$ and take the inner product with $\boldsymbol{v}$, $G_{\boldsymbol{w}}(\boldsymbol{v})$ is the maximum value of this inner

product considering all permutations of $\boldsymbol{w}$. Formally, $G_{\boldsymbol{w}}(\boldsymbol{v}) = \max_{\boldsymbol{w}_\tau \in \Pi} \boldsymbol{w}_\tau^\top \boldsymbol{v}$ where Π is the set of all permutations (of the coefficients) of the vector $\boldsymbol{w}$. Thus, with $\boldsymbol{v}$ fixed, $G_{\boldsymbol{w}}(\boldsymbol{v})$ can be computed by the following LP problem:

$$\max \sum_{i\in[n]} \sum_{k\in[n]} p_{ik} w_i v_k \tag{4.13a}$$

$$\text{s.t} \sum_{i\in[n]} p_{ik} = 1 \qquad \forall k \in [n] \tag{4.13b}$$

$$\sum_{k\in[n]} p_{ik} = 1 \qquad \forall i \in [n] \tag{4.13c}$$

$$p_{ik} \geq 0 \qquad \forall i,k \in [n]. \tag{4.13d}$$

Take a dual of (4.13) and integrate it into (OWA$-\mathcal{P}$), we get the second MILP formulation for (OWA$-\mathcal{P}$), called C-MILP:

$$\text{(C-MILP)} \quad \begin{aligned} &\min \sum_{i\in[n]} (\alpha_i + \beta_i) && (4.14a)\\ &\text{s.t } \alpha_i + \beta_k \geq w_i v_k \quad \forall i,k \in [n] && (4.14b)\\ &\boldsymbol{v} = \boldsymbol{C}\boldsymbol{x} && (4.14c)\\ &\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} && (4.14d)\\ &\boldsymbol{x} \in \{0,1\}^m && (4.14e) \end{aligned}$$

We also consider a dual of C-MILP's continuous relaxation (denoted as $(\mathcal{D}_C)$) as the previous section.

$$(\mathcal{D}_C) \quad \begin{aligned} &\max \; -\boldsymbol{u}^T\boldsymbol{b} - \sum_{k\in[m]} t_k && (4.15a)\\ &\text{s.t.} \sum_{i\in[n]} p_{ik} w_i + z_k = 0 \qquad \forall k \in [n] && (4.15b)\\ &(\boldsymbol{u}\boldsymbol{A})_k - (\boldsymbol{z}\boldsymbol{C})_k + t_k \geq 0 \quad \forall k \in [m] && (4.15c)\\ &\sum_{i\in[n]} p_{ik} = 1 \qquad \forall k \in [n] && (4.15d)\\ &\sum_{k\in[n]} p_{ik} = 1 \qquad \forall i \in [n] && (4.15e)\\ &p_{ik} \geq 0 \qquad \forall i,k \in [n] && (4.15f)\\ &t_k \geq 0 \qquad \forall k \in [m] && (4.15g)\\ &u_k \geq 0 \qquad \forall k \in [K]. && (4.15h) \end{aligned}$$

Then, a dual of the above formulation with $\boldsymbol{p}$ fixed is:

$$(RP_{\boldsymbol{p}}) \quad \begin{aligned} &\min \sum_{i\in[n]} \left(\sum_{k\in[n]} w_k p_{ik} \right) v_i && (4.16a)\\ &\text{s.t. } \boldsymbol{v} = \boldsymbol{C}\boldsymbol{x} && (4.16b)\\ &\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} && (4.16c)\\ &\boldsymbol{x} \in [0,1]^m && (4.16d) \end{aligned}$$

We denote $(P_{\boldsymbol{p}})$ the integer version of $(RP_{\boldsymbol{p}})$ over $\boldsymbol{x}$ (i.e., the constraint (4.16d) is replaced by $\boldsymbol{x} \in \{0,1\}^m$). Similar to O-MILP, solving $(RP_{\boldsymbol{p}})$ with $\boldsymbol{x}$ discrete yields a feasible solution for C-MILP.

In comparison, C-MILP's size is smaller than O-MILP's due to the number of extra variables. More precisely, although both formulations use the same number of additional constraints (i.e., n^2), C-MILP utilizes only $2n$ new variables instead of $n^2 + n$ as in O-MILP. These formulations were compared experimentally for several continuous optimization problems [2]. In this paper, we provide a theoretical analysis (Section 4.3) in order to understand these formulations better, and we experimentally evaluate them for combinatorial optimization (Section 4.5).

4.3 Theoretical Analysis

In this section, we show that the two formulations are equivalent in terms of linear relaxations. Then, we give an estimation of the GGI value corresponding to optimal solutions of (Min$-\mathcal{P}$) based on that of (OWA$-\mathcal{P}$).

4.3.1 Relation between the formulations

We first introduce two notations. Let $\mathcal{Y}$ be the set of points that satisfy (4.11b) and (4.11c), and $\mathcal{P}$ be the set of points that satisfy (4.15d), (4.15e) and (4.15f).

The following theorem relates the problems $(P_{\boldsymbol{y}})$ and $(P_{\boldsymbol{p}})$.

Theorem 4.3.1. *There exists a one-to-one correspondence $\phi : \mathcal{P} \to \mathcal{Y}$ such that $\forall \boldsymbol{p} \in \mathcal{P}$, problems $(P_{\boldsymbol{p}})$ and $(P_{\phi(\boldsymbol{p})})$ have the same solutions.*

This theorem directly follows from the following two lemmas, which show that the set of feasible solutions of $(P_{\boldsymbol{p}})$ for $\boldsymbol{p} \in \mathcal{P}$ and that for $(P_{\boldsymbol{y}})$ for $\boldsymbol{y} \in \mathcal{Y}$ coincide.

Lemma 4.3.2. $\forall \boldsymbol{p}^* \in \mathcal{P}$, $\exists \boldsymbol{y}^* \in \mathcal{Y}$ *such that* $\forall i \in [n], \sum_{k\in[n]} w_k p^*_{ik} = \sum_{k\in[n]} y^*_{ik}$.

Proof. It is sufficient to prove the result for the extreme points of polytope $\mathcal{P}$. As any other point $\boldsymbol{p'}$ is a convex combination of some extreme solutions $\boldsymbol{p^1}, \ldots, \boldsymbol{p^k}$, its counterpart $\boldsymbol{y'}$ can be obtained by the same convex combination of the counterparts $\boldsymbol{y^1}, \ldots, \boldsymbol{y^k}$ of $\boldsymbol{p^1}, \ldots, \boldsymbol{p^k}$.

Recall that the set of extreme points of $\mathcal{P}$ is exactly the set of permutations on set $[n]$. Let $\boldsymbol{p}^*$ be an extreme point of $\mathcal{P}$. Therefore, all the components of $\boldsymbol{p}^*$ are null except for some n components $p^*_{i_1 1} = p^*_{i_2 2} = \ldots = p^*_{i_n n} = 1$. The counterpart $\boldsymbol{y}^*$ of $\boldsymbol{p}^*$ can be built as follows:

$$\begin{pmatrix} y^*_{i_1 1} = w'_1 & y^*_{i_1 2} = w'_2 & \ldots & \ldots & \ldots & y^*_{i_1 n} = w'_n \\ y^*_{i_2 1} = 0 & y^*_{i_2 2} = w'_2 & \ldots & \ldots & \ldots & y^*_{i_2 n} = w'_n \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ y^*_{i_k 1} = 0 & y^*_{i_k 2} = 0 & \ldots & y^*_{i_k k} = w'_k & \ldots & y^*_{i_k n} = w'_n \\ \ldots & \ldots & \ldots & \ldots & \ldots & y^*_{i_n n} = w'_n \end{pmatrix}$$

By considering any column $k \in [n]$, one can check: $\boldsymbol{y}^* \in \mathcal{Y}$, i.e., $\sum_{j\in[n]} y^*_{i_j k} = k w'_k$ and $y^*_{i_j k} \leq w'_k$. Moreover, by summing any row i_j for $j \in [n]$, one can check: $\sum_{k\in[n]} y^*_{i_j k} = w_j = \sum_{k\in[n]} w_k p^*_{i_j k}$. □

Lemma 4.3.3. $\forall \boldsymbol{y}^* \in \mathcal{Y}$, $\exists \boldsymbol{p}^* \in \mathcal{P}$ *such that* $\forall i \in [n], \sum_{k\in[n]} w_k p^*_{ik} = \sum_{k\in[n]} y^*_{ik}$.

Proof. Similarly to Lemma 4.3.2, it suffices to show the result for any extreme solution $\boldsymbol{y}$ in $\mathcal{Y}$. Let us consider the following transportation problem (TP) where the set of the supply nodes and the set of demand nodes are both $[n]$. For any $i, j \in [n]$, the supply at supply node i is equal to $\sum_{k\in[n]} y^*_{ik}$ and the demand of demand node j is equal to w_j. Note that the total supply is equal to the total demand, because $\sum_{i\in[n]} y^*_{ik} = kw'_k$, $\sum_{k\in[n]}\sum_{i\in[n]} y^*_{ik} = \sum_{k\in[n]} kw'_k = \sum_{i\in[n]} w_i$. For any feasible solution of TP, $\boldsymbol{\Phi} = \left(\Phi_{ik}\right)_{i,k\in[n]}$ where Φ_{ik} is the value of the commodity transported from supply node i to demand node k, we can define a vector $\boldsymbol{p}$ as follows: $p_{ik} = \frac{\Phi_{ik}}{w_k}$ for all $i, k \in [n]$, which by definition satisfies:

$$\sum_{i\in[n]} p_{ik} = \sum_{i\in[n]} \frac{\Phi_{ik}}{w_k} = \frac{\sum_{i\in[n]} \Phi_{ik}}{w_k} = 1, \tag{4.17}$$

and

$$\sum_{k\in[n]} y^*_{ik} = \sum_{k\in[n]} \Phi_{ik} = \sum_{k\in[n]} w_k p_{ik}. \tag{4.18}$$

If $\boldsymbol{p}$ does not satisfy (*) $\sum_{k\in[n]} p_{ik} = 1$ for all $i \in [n]$, we will construct a sequence $\boldsymbol{\Phi}^t$ and its associated $\boldsymbol{p}^t$ such that $||\sum_{k\in[n]} \boldsymbol{p}^t_{.k} - \mathbf{1}||_1 = \sum_{i\in[n]} |\sum_{k\in[n]} p^t_{ik} - 1|$ tends to zero, where $\mathbf{1}$ denotes the vector in $\mathbb{R}^n$ whose components are all equal to 1.

We first show that if $\boldsymbol{p}$ does not satisfy (*), then $\exists i, j, h, l \in [n]$ such that $h < l$ and $\Phi_{il} > 0$ and $\Phi_{jh} > 0$. Let $P_1 = \{i \in [n] \mid \sum_{k\in[n]} p^*_{ik} \geq 1\}$ and $P_2 = \{i \in [n] \mid \sum_{k\in[n]} p^*_{ik} < 1\}$ be a partition of $[n]$. By assumption, $P_2 \neq \emptyset$. Let $p_1 = |P_1|$, $p_2 = |P_2|$, $W_1 = [p_1]$, and $W_2 = \{p_1 + 1, p_1 + 2, \ldots, n\}$. Among the following four cases, only one is possible:

- There exist both strictly positive flows from supply nodes in P_1 to demand nodes in W_2 and strictly positive flows from supply nodes in P_2 to demand nodes in W_1. Thus, there are some $i \in P_1$, $j \in P_2$, $h \in W_1$ and $l \in W_2$ such that $\Phi^*_{il} > 0$ and $\Phi^*_{jh} > 0$.

- There exist strictly positive flows from supply nodes in P_1 to demand nodes in W_2 but no strictly positive flows from supply nodes in P_2 to demand nodes in W_1. Observe that the p_1 greatest values that $\sum_{k\in[n]} y^*_{ik}$ for $i \in [n]$ can take are $w'_1 + w'_2 + \ldots + w'_{n-1} + w'_n = w_1$, $w'_2 + w'_3 + \ldots + w'_n = w_2$, …, $w'_{p_1} + w'_{p_1+1} + \ldots + w'_n = w_{p_1}$. Hence $\sum_{i\in P_1}\sum_{k\in[n]} y^*_{ik} \leq \sum_{i\in[p_1]} w_i$, i.e., the total supply in P_1 is less than or equal to the total demand in W_1. Thus, it is impossible that W_1 only receive flows from P_1 and P_1 can still send positive flows to W_2.

- There exist strictly positive flows from supply nodes in P_2 to demand nodes in W_1 but no strictly positive flows from supply nodes in P_1 to demand nodes in W_2. Thus, strictly positive flows to W_2 come uniquely from P_2. Hence, we can observe that

$$\sum_{i\in P_2, k\in W_2} p^*_{ik} = \sum_{i\in[n], k\in W_2} \frac{\Phi^*_{ik}}{w_k} = \sum_{k\in W_2} \frac{w_k}{w_k} = |W_2| = p_2$$

  But as $\sum_{k\in[n]} p^*_{ik} < 1$ for all $i \in P_2$, $\sum_{i\in P_2, k\in[n]} p^*_{ik} < |P_2| = p_2$, we have then

$$p_2 = \sum_{i\in P_2, k\in W_2} p^*_{ik} \leq \sum_{i\in P_2, k\in[n]} p^*_{ik} < p_2,$$

  which is contradictory.

- There are no strictly positive flows from supply nodes in P_2 to demand nodes in W_1 and no strictly positive flows from supply nodes in P_1 to demand nodes in W_2. This case is impossible due to the same reason as the previous one.

Let $\mathbf{\Phi}^0$ be a feasible solution of TP and $\boldsymbol{p}^0$ its associated vector. For any $t \in \mathbb{N}$, two cases can occur: (1) if $\boldsymbol{p}^t$ satisfies (*), $\mathbf{\Phi}^{t+1} = \mathbf{\Phi}^t$ and $\boldsymbol{p}^{t+1} = \boldsymbol{p}^t$; otherwise (2) $\mathbf{\Phi}^{t+1}$ and $\boldsymbol{p}^{t+1}$ can defined according to the following procedure. As (*) is not satisfied, there exist $i, j, h, l \in [n]$ such that $h < l$ and $\Phi^t_{il} > 0$ and $\Phi^t_{jh} > 0$. We then define $\Phi^{t+1} = \Phi^t$ except for the following terms: $\Phi^{t+1}_{ih} = \Phi^t_{ih} + \epsilon$, $\Phi^{t+1}_{il} = \Phi^t_{il} - \epsilon$, $\Phi^{t+1}_{jh} = \Phi^t_{jh} - \epsilon$ and $\Phi^{t+1}_{jl} = \Phi^t_{jl} + \epsilon$, where $\epsilon = \min(\Phi^t_{il}, \Phi^t_{jh}, \frac{(\sum_{k\in[n]} p^t_{ik} - 1)w_l w_h}{w_h - w_l}, \frac{(1-\sum_{k\in[n]} p^t_{jk})w_l w_h}{w_h - w_l})$. Note that $\epsilon > 0$ since $w_h > w_l$.

Consequently, $\boldsymbol{p}^{t+1}$ satisfies: $p^{t+1}_{ih} = p^t_{ih} + \frac{\epsilon}{w_h}$, $p^{t+1}_{il} = p^t_{il} - \frac{\epsilon}{w_l}$, $p^{t+1}_{jh} = p^t_{jh} - \frac{\epsilon}{w_h}$ and $p^{t+1}_{jk} = p^t_{jl} + \frac{\epsilon}{w_l}$. By construction, $\mathbf{\Phi}^{t+1}$ and $\boldsymbol{p}^{t+1}$ verify (4.17) and (4.18). Moreover, $\sum_{k\in[n]} p^{t+1}_{ik}$ is decreased by $\frac{\epsilon(w_l - w_h)}{w_h w_l}$ and the sum $\sum_{k\in[n]} p^*_{jk}$ is increased by the same quantity. Hence, $||\sum_{k\in[n]} \boldsymbol{p}^t_{\cdot k} - \mathbf{1}||_1 > ||\sum_{k\in[n]} \boldsymbol{p}^{t+1}_{\cdot k} - \mathbf{1}||_1$. Clearly, sequence $(\boldsymbol{p}^t)$ converges to a vector in $\mathcal{P}$, which proves this lemma. □

4.3.2 Quality estimation for the optimal solution of (Min$-\mathcal{P}$)

Due to the size of the formulations, MILP solvers can only solve small-sized instances of OWA combinatorial optimization within a reasonable amount of time. Thus, if the optimal solution of (Min$-\mathcal{P}$) is "good enough" for (OWA$-\mathcal{P}$), solving exactly large-sized instances is needless. The following theorem provides an estimation to evaluate the quality of (Min$-\mathcal{P}$)'s optimal solution regarding the GGI value.

Theorem 4.3.4. *Assume that there exists an approximation ratio r between (Min$-\mathcal{P}$) and its continuous relaxation (Min $-\ \mathcal{RP}$). Let $(\overline{\boldsymbol{x}}, \overline{\boldsymbol{v}})$ be an optimal solution to (Min$-\mathcal{P}$) and $C = r \min\left(\frac{nw_1}{\sum_{i=1}^n w_i}, \frac{n\theta_1(\overline{\boldsymbol{v}})}{\sum_{i\in[n]} \overline{v}_i}\right)$, we have:*

$$G_{\boldsymbol{w}}(\overline{\boldsymbol{v}}) \leq C \times G_{\boldsymbol{w}}(\boldsymbol{v}^*) \tag{4.19}$$

where $(\boldsymbol{x}^*, \boldsymbol{v}^*)$ is an optimal solution to (OWA$-\mathcal{P}$).

Proof. In this proof, we will use $OPT(\mathcal{P})$ to denote the optimal objective value of problem $\mathcal{P}$ (for example, $OPT(\mathcal{D}_O)$ is the optimal objective value of $(\mathcal{D}_O)$). From the assumption, we have:

$$OPT(\text{Min} - \mathcal{P}) \leq rOPT(\text{Min} - \mathcal{RP}) \tag{4.20}$$

To establish the result (4.19), we only need to prove that:

$$G_{\boldsymbol{w}}(\overline{\boldsymbol{v}}) \leq rC_i \times G_{\boldsymbol{w}}(\boldsymbol{v}^*), \qquad \forall i = 1, 2$$

where $C_1 = \dfrac{nw_1}{\sum_{i=1}^n w_i}$, $C_2 = \dfrac{n\theta_1(\overline{\boldsymbol{v}})}{\sum_{i\in[n]} \overline{v}_i}$.

The bound with C_1 results from C-MILP while that of C_2 comes from O-MILP.

1. Proof of $G_{\boldsymbol{w}}(\overline{\boldsymbol{v}}) \leq rC_1 \times G_{\boldsymbol{w}}(\boldsymbol{v}^*)$: Let $\overline{\boldsymbol{p}} \in \mathbb{R}^{n\times n}$ subject to $\overline{p}_{ik} = 1/n$ for $i, k \in [n]$. Obviously, $\overline{\boldsymbol{p}}$ satisfies constraints (4.15d) - (4.15f) of $(\mathcal{D}_C)$. The objective value of $(RP_{\overline{\boldsymbol{p}}})$ is:

$$\sum_{i\in[n]} \left(\sum_{j\in[n]} w_j \overline{p}_{ij}\right) v_i = \frac{\sum_{j\in[n]} w_j}{n} \sum_{i\in[n]} v_i \tag{4.21}$$

Therefore, $OPT(P_{\overline{\boldsymbol{p}}}) = \frac{\sum_{j\in[n]} w_j}{n} OPT(\text{Min} - \mathcal{P})$ and $OPT(RP_{\overline{\boldsymbol{p}}}) = \frac{\sum_{j\in[n]} w_j}{n} OPT(\text{Min} - \mathcal{RP})$. This observation leads to:

$$\begin{aligned} G_{\boldsymbol{w}}(\overline{\boldsymbol{v}}) &= \sum_{i\in[n]} w_i \theta_i(\overline{\boldsymbol{v}}) \\ &= \sum_{i\in[n]} \frac{n w_i}{\sum_{j\in[n]} w_j} \frac{\sum_{j\in[n]} w_j}{n} \theta_i(\overline{\boldsymbol{v}}) \\ &\leq \frac{n w_1}{\sum_{j\in[n]} w_j} \left(\sum_{i\in[n]} \frac{\sum_{j\in[n]} w_j}{n} \theta_i(\overline{\boldsymbol{v}}) \right) \\ &= C_1 \frac{\sum_{j\in[n]} w_j}{n} \sum_{i\in[n]} \overline{v}_i \\ &= C_1 \frac{\sum_{j\in[n]} w_j}{n} OPT(\text{Min} - \mathcal{P}) \end{aligned} \tag{4.22}$$

Using the relation between $OPT(\text{Min} - \mathcal{P})$ and $OPT(\text{Min} - \mathcal{RP})$, we get:

$$\begin{aligned} &\frac{\sum_{j\in[n]} w_j}{n} OPT(\text{Min} - \mathcal{P}) \\ &\leq r \frac{\sum_{j\in[n]} w_j}{n} OPT(\text{Min} - \mathcal{RP}) \\ &= r OPT(RP_{\overline{\boldsymbol{p}}}) \\ &\overset{(a)}{\leq} r OPT(\mathcal{D}_C) \\ &\overset{(b)}{\leq} r OPT(\text{C-MILP}) \\ &= r G_{\boldsymbol{w}}(\boldsymbol{v}^*) \end{aligned} \tag{4.23}$$

where (a) : when *fixing* $\boldsymbol{p} = \overline{\boldsymbol{p}}$, we have $OPT(\mathcal{D}_C) = OPT(RP_{\overline{\boldsymbol{p}}})$ (strong duality) and both are smaller than $OPT(\mathcal{D}_C)$ without fixing $\boldsymbol{p} = \overline{\boldsymbol{p}}$; (b): since $(\mathcal{D}_C)$ is a dual of C-MILP's continuous relaxation.

We obtain the proof by combining (4.22) and (4.23).

2. Proof of $G_{\boldsymbol{w}}(\overline{\boldsymbol{v}}) \leq rC_2 \times G_{\boldsymbol{w}}(\boldsymbol{v}^*)$: Recall that $w'_i = w_i - w_{i+1}$ for $i \in [n-1]$ and $w'_n = w_n$. From O-MILP, we have:

$$\begin{aligned} G_{\boldsymbol{w}}(\overline{\boldsymbol{v}}) &= \sum_{k\in[n]} \left(\sum_{i\in[k]} \theta_i(\overline{\boldsymbol{v}}) \right) w'_k \\ &= \sum_{k\in[n]} \frac{n \sum_{i\in[k]} \theta_i(\overline{\boldsymbol{v}})}{k \sum_{i\in[n]} \overline{v}_i} \left(\frac{k}{n} \sum_{i\in[n]} \overline{v}_i \right) w'_k \\ &\leq \frac{n\theta_1(\overline{\boldsymbol{v}})}{\sum_{i\in[n]} \overline{v}_i} \sum_{k\in[n]} \left(\frac{k}{n} \sum_{i\in[n]} \overline{v}_i \right) w'_k \\ &= C_2 \left(\sum_{k\in[n]} \frac{k}{n} w'_k \right) \sum_{i\in[n]} \overline{v}_i \\ &= C_2 \frac{\sum_{j\in[n]} w_j}{n} \sum_{i\in[n]} \overline{v}_i \\ &= C_2 \frac{\sum_{j\in[n]} w_j}{n} OPT(\text{Min} - \mathcal{P}) \end{aligned} \tag{4.24}$$

The desired result follows by the application of (4.23) and (4.24).

□

Remark 4.3.5. Compared to the ratio established in [8], Theorem 4.3.4 generally has an additional factor r. This factor depends on the actual problem. For instance, if one considers the metric TSP where edge weights satisfy triangle inequalities, this factor will be $3/2$ [62].

4.4 A Primal-Dual Heuristic

Due to the number of additional variables and constraints, the MILP formulations are only efficient for small-sized instances. A primal-dual heuristic based on O-MILP is proposed in [8] to deal with larger-sized instances. In this section, we generalize this method for both formulations, sketched in Algorithm 11.

Algorithm 11

Input: $(\boldsymbol{C}, \boldsymbol{A}, \boldsymbol{b})$.
Output: A solution to $(\text{OWA} - \mathcal{P})$

```
initialize y^(0) (resp. p^(0))
repeat
    t ← t + 1
    solve (P_{y^(t-1)}) (resp. (P_{p^(t-1)})) to obtain a feasible solution (v^(t), x^(t))
    update y^(t) (resp. p^(t)) based on y^(t-1) (resp. p^(t-1)) and (v^(t), x^(t))
until max iteration has been reached or change on y^(t) (resp. p^(t)) is small
return (v^(t), x^(t)) with smallest GGI value
```

The algorithm starts from an initialization of $\boldsymbol{y}^{(0)}$ (resp. $\boldsymbol{p}^{(0)}$) satisfying conditions (4.11b), (4.11c), and (4.11f) (resp. (4.15d) - (4.15f)). For example, we can choose $\boldsymbol{y}^{(0)}$ (resp. $\boldsymbol{p}^{(0)}$) as proposed in the proof of Theorem 4.3.4. Then $\boldsymbol{y}^{(t)}$ (resp. $\boldsymbol{p}^{(t)}$) is updated iteratively based on the improvement of lower bounds obtained from the Lagrangian relaxation corresponding to O-MILP (resp. C-MILP). For space reasons, we focus on formulation C-MILP (formulation O-MILP was presented in [8]). In detail, the Lagrangian relaxation of C-MILP with respect to constraint (4.14b) can be defined as follows

$$\begin{aligned}\mathcal{L}(\boldsymbol{\lambda}) = \min & \sum_{i\in[n]}(1-\sum_{j\in[n]}\lambda_{ij})\alpha_i + \sum_{j\in[n]}(1-\sum_{i\in[n]}\lambda_{ij})\beta_i \\ & + \sum_{i\in[n]}\sum_{k\in[n]}\lambda_{ik}w_i v_k \\ \text{s.t. } & \boldsymbol{v} = \boldsymbol{C}\boldsymbol{x} && \text{(4.25a)} \\ & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} && \text{(4.25b)} \\ & \boldsymbol{x} \in \{0,1\}^m && \text{(4.25c)}\end{aligned}$$

where $\boldsymbol{\lambda} = (\lambda_{ik})_{i\in[n],k\in[n]}$ is a Lagrangian multiplier.

Instance	O-MILP		C-MILP		$\mathcal{H}_O$		$\mathcal{H}_C$	
	CPU1(s)	CPU2(s)	CPU1(s)	CPU2 (s)	CPU(s)	Gap(%)	CPU(s)	Gap(%)
burma14	0.83	0.27	0.45	0.20	1.50	0.23%	1.62	0.06%
gr17	1.68	0.08	1.28	0.07	1.97	1.15%	.2.32	0.23%
gr21	1.46	1.40	1.08	1.04	1.82	0.02%	2.25	0.00%
gr24	8.47	0.30	8.46	0.55	3.59	0.00%	4.09	0.00%
fri26	21.01	9.35	49.97	9.08	5.74	0.00%	4.12	0.02%
bayg29	31.06	0.70	35.86	0.55	9.31	2.51%	8.38	1.11%
bays29	37.89	7.44	57.85	21.06	9.92	0.86%	11.10	1.19%
swiss42	354.47	2.23	915.22	1.38	16.86	0.51%	20.77	0.50%
att48	731.25	18.89	655.06	4.52	45.16	1.77%	97.09	1.66%
gr48	863.02	334.84	2419.39	354.91	57.17	2.86%	56.19	1.75%
hk48	1943.39	440.99	719.89	443.49	32.21	0.03%	31.14	0.48%
brazil58	10802.1*	180.27	10801.6*	664.29	21.85	(4770.24)	21.88	(4772.03)
st70	10803.0*	10802.7*	10801.4*	10800.3*	207.48	(35.89)	169.53	(35.97)
kroA100	10800.1*	10800.0*	10803.8*	10800.5*	1202.54	(811.42)	1657.44	(790.76)

Table 4.1. Numerical results for OWA TSP.

Solving (4.25) obtains a lower bound for C-MILP. To get a meaningful bound, the Lagrangian multiplier $\boldsymbol{\lambda}$ has to belong to set $\mathbb{L} = \{\boldsymbol{\lambda} \in \mathbb{R}_+^{n\times n} | \sum_{i\in[n]} \lambda_{ik} = 1 \; \forall k \in [n], \sum_{k\in[n]} \lambda_{ik} = 1 \; \forall i \in [n]\}$. Interestingly, if we decompose formulation (4.25) into two subproblems $(S1)$, $(S2)$ where $(S1)$ over $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ and $(S2)$ over $(\boldsymbol{v}, \boldsymbol{x})$, the subproblem (S_2) is exactly formulation $(P_{\boldsymbol{p}})$. Thus, updating $\boldsymbol{p}$ is equivalent to updating a projected sub-gradient step, i.e:

$$\lambda'_{ij} \leftarrow \lambda_{ij} - \gamma(\alpha_i + \beta_j - w_i v_j) \qquad \forall i \in [n], k \in [n] \tag{4.26a}$$

$$\boldsymbol{\lambda} \leftarrow \arg\min_{\boldsymbol{\lambda}\in\mathbb{L}} ||\boldsymbol{\lambda}' - \boldsymbol{\lambda}|| \tag{4.26b}$$

where γ is the learning rate. Since the constraints of $\mathbb{L}$ involve both rows and columns of $\boldsymbol{\lambda} \in \mathbb{R}^{n\times n}$, the projection on $\mathbb{L}$ cannot be solved by the capped simplex projection as in [8]. Hence, we perform the projection on $\mathbb{L}$ by minimizing a convex quadratic function over linear constraints.

4.5 Numerical results

In this section, we provide experimental results of two MILP formulations and the primal-dual heuristic on an OWA combinatorial optimization problem, i.e., the OWA TSP. The testbed consists of TSPLIB instances [46] with the number of nodes in the range of 14 to 100. The weight $\boldsymbol{w}$ of the OWA is defined by $w_k = 1/k^2$ for $k \in [n]$. We limited the solving time for each instance to three hours (10800s). All experiments are implemented in C++ programming language and conducted on a computer with 3GHz Intel Core i5 CPU and 16GB of RAM. We used ILOG CPLEX version 12.10.0 with default parameters and one thread to solve LP and MILP problems.

In (OWA-TSP), since the number of subtour elimination constraints is exponential, we generate them progressively during the branch-and-cut algorithm. In detail, when a feasible solution $(\boldsymbol{r}^*, \boldsymbol{d}^*, \boldsymbol{x}^*, \boldsymbol{v}^*)$ (resp. $(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*, \boldsymbol{x}^*, \boldsymbol{v}^*)$) of O-MILP (resp. C-MILP) is found, we construct a

graph $G^* = (V, E^*)$ where $E^* = \{ij \in E^d | x^*_{ij} = 1\}$. Then, a strongly connected component that does not contain all nodes of G^* corresponds to a subtour elimination constraint violated by $\boldsymbol{x}^*$. Violated constraints (if any) are added to the formulations.

For simplicity, let $\mathcal{H}_O$, $\mathcal{H}_C$ be Algorithm 11 utilizing O-MILP and C-MILP, respectively. We initialized $y^{(0)}_{ik} = \frac{k}{n} w'_k$ and $p^{(0)}_{ik} = \frac{1}{n}$ for $i, k \in [n]$. At time step t, the learning rate $\gamma^{(t)}$ is computed as $\gamma^{(t)} = \frac{OPT(P) - G_{\boldsymbol{w}}(v')}{||\boldsymbol{\lambda}^{(t-1)}||_2}$ where $OPT(P)$ is the objective value of $(P_{\boldsymbol{y}^{(t-1)}})$ (resp. $(P_{\boldsymbol{p}^{(t-1)}})$), $\boldsymbol{v}'$ is the best feasible solution of O-MILP (resp. C-MILP) so far.

In Table 4.1, the number in the instance's name is the number of nodes. Columns "O-MILP" and "C-MILP" regroup results of O-MILP and C-MILP, where subcolumn "CPU1" reports the time in seconds that CPLEX spent to obtain the optimal solution and subcolumn "CPU2" reports the time to obtain a feasible solution as small as the solution of $\mathcal{H}_O$ (resp. $\mathcal{H}_C$). Results of instances that can not be solved within the time limit (i.e., 10800s) are marked with an asterisk (*). Columns "$\mathcal{H}_O$" and '$\mathcal{H}_C$" report respectively results of $\mathcal{H}_O$ and $\mathcal{H}_C$, where subcolumn "CPU" is the runtime and "Gap/Obj" provides the gap in percentage between the solutions of O-MILP (resp. C-MILP) and Algorithm 11. In the case that MILP formulations can not be solved within the time limit, we provide instead the objective value (numbers in parentheses, without the percent sign) corresponding to a solution found by $\mathcal{H}_O$ (resp. $\mathcal{H}_C$).

Numerical results confirm that MILP formulations can only handle small-sized instances. The running time spent to solve these formulations increases rapidly with instances' size and can reach up to around three hours. For example, instances with more than 50 vertices can not be solved within three hours. Interestingly, although C-MILP was shown to be better than O-MILP for OWA simplified portfolio optimization problem [2], its performance is worse than O-MILP's one when applying for OWA TSP. In contrast, $\mathcal{H}_O$ and $\mathcal{H}_C$ solve instance quickly, especially large-sized instances. The time spent by Algorithm 11 increases acceptably with instance size. Furthermore, their solutions are high-quality, namely that their optimality gap is at most 3%.

4.6 Conclusion

In this chapter, we study OWA combinatorial optimization, a fair variant of combinatorial optimization that uses the GGI objective function. Our result extends the previous one [8] to deal with OWA combinatorial optimization. Our theoretical results show that two proposed schemes of OWA linearization in the literature are equivalent in terms of linear relaxations in the context of OWA combinatorial optimization. These linearizations can also be exploited to compare the GGI values of optimal solutions of the original and fair version of a combinatorial optimization problem. Numerical results show that O-MILP can be solved faster despite using more variables than C-MILP for OWA TSP. Our future work will focus on deriving more efficient heuristics and developing theoretical guarantees for those heuristics.

Part II

Machine Learning for Combinatorial Optimization

Chapter 5

Machine learning to accelerate branch-and-cut for combinatorial optimization

Summary

In this chapter, we explore using machine learning to improve branch-and-cut algorithms for solving combinatorial optimization problems. Over the past decade, machine learning has increasingly replaced human-engineered algorithms for many tasks. With the rise of deep learning, machine learning models can now learn end-to-end without requiring hand-crafted feature engineering or multi-stage pipelines. By learning from big data, deep learning has surpassed classic methods in many domains, including computer vision, natural language processing, speech processing, and recommendation systems. Furthermore, combining deep learning with RL can produce more powerful models than human-designed heuristics. A prime example is AlphaGo Zero [63], a deep RL-based algorithm for playing the board game Go. Although only trained on a small set of tremendous search space, AlphaGo Zero still can defeat the world's best human players.

Inspired by such successes, we believe machine learning has similar potential to replace hand-crafted heuristics for the internal decisions of branch-and-cut. Even minor improvements in optimization performance could bring tremendous savings when solving an extensive number of problems daily.

Previous work has shown promise in using machine learning to improve fundamental decision strategies of branch-and-cut, including variable selection [21, 22, 64], node selection [23], heuristic selection [65, 66], and cut selection [24, 67, 68]. However, *cut generation*—a critical decision problem that arises when combining branch-and-bound and cutting planes in branch-and-cut—has received less attention despite its importance. In particular, consider an MILP formulation and suppose that we just solved one of its LP relaxations at a node of the enumeration tree and obtained an optimal solution $\boldsymbol{x}^{\text{LP}}$ to this relaxation. If $\boldsymbol{x}^{\text{LP}}$ is fractional for at least one variable that must take on an integer value in the MILP, what strategy should be employed next? Should we generate additional cuts to tighten the LP relaxation or branch on the fractional variables? There is a trade-off: generating cuts can strengthen the formulation but slowdowns processing the node due to separation process and grows the LP relaxation size; branching is faster to execute but exponentially grows the enumeration tree, potentially increasing the overall solving time. We should prioritize choices that lead to faster pruning of the branch-and-bound tree: infeasibility or integrality of the created subproblems or pruning by bounds. Nevertheless, to the best of our knowledge, no theory can directly measure the impact of these choices, which can usually only be observed after the algorithm has finished running. Thus, deciding whether to cut or branch poses a non-trivial challenge when designing branch-and-cut algorithms.

The majority of previous research on cut-related strategies in branch-and-cut has focused on general-purpose rather than problem-specific cuts. However, most combinatorial optimization problems have underlying structures that can be exploited to generate *"strong"* valid inequalities, called *combinatorial cuts*, which are typically facet-defining for the problem's polytope. Combinatorial cuts can considerably strengthen the LP relaxation and eliminate a substantial portion of the infeasible region; thus, they play a critical role in solving combinatorial problems of meaningful size. Numerous studies in the literature have proposed various classes of such cuts for specific optimization problems, yielding significant enhancements in branch-and-cut performance. The typical approach of these studies is to identify facet-defining inequalities of the convex hull of feasible solutions, present separation algorithms to generate them,

and demonstrate computational gains. However, little attention has been paid to the practical implementation challenges of combinatorial cuts. For example, the corresponding separation routine of these cuts is usually computationally expensive. Naively generating combinatorial cuts can make the separation process the dominant task and hurt the overall performance. Furthermore, extra cuts may sometimes not strengthen the LP relaxation, rendering separation wasteful. Hence, it requires careful consideration when generating combinatorial cuts during branch-and-cut.

Motivated by these research gaps, in this chapter, we develop the first machine learning framework for learning strategies to generate combinatorial cuts in branch-and-cut. Given a combinatorial optimization problem and an associated class of combinatorial cuts, our framework learns a policy for selecting between generating cuts or branching to perform at enumeration tree nodes in order to minimize overall running time. The proposed framework contains two components: 1) A *cut detector* to detect whether valid cuts exist, and 2) A *cut evaluator* to decide whether generating them is worthwhile. The models are trained on small instances using a combination of supervised and reinforcement learning. They can then be applied to larger instances.

To evaluate the effectiveness of the proposed framework, we implement it for two well-known NP-hard combinatorial optimization problems: the TSP with subtour elimination constraints and the Max-Cut problem with cycle inequalities. Experiments show that the learned cut generation policies substantially accelerate branch-and-cut, even on instances of different sizes from its training counterparts. These results help fulfill the potential of using machine learning to enhance branch-and-cut inner strategies.

The remainder of this chapter is organized as follows. Section 5.1 presents the cut generation problem in branch-and-cut and empirically shows its impact on performance. Section 5.2 overviews previous works about cut generation and machine learning for branch-and-cut. Section 5.3 introduces our general framework for learning a cut generation policy. Sections 5.4 and 5.5 respectively present in detail the framework's components: the cut detector and cut evaluator. Computational results to evaluate the method's effectiveness are provided in Section 5.6. Finally, some conclusions and limitations of the approaches are discussed in Section 5.7.

5.1 Cut generation problem

Given a combinatorial optimization problem $\mathcal{P}$ and a cut class $\mathcal{C}$ associated with $\mathcal{P}$, one of the primary decisions in branch-and-cut for solving $\mathcal{P}$ is whether to generate cuts of type $\mathcal{C}$ or branch on tree nodes. This choice significantly impacts overall performance.

On the one hand, generating cuts can tighten linear relaxations and improve bounds to prune the branch-and-bound tree faster. However, cut generation can also worsen performance in certain situations. Firstly, the separation routine for $\mathcal{C}$-type cuts may be computationally expensive, especially for large instances. Additionally, some executions may fail to produce valid cuts, which can result in a waste of time. Finally, at some tree nodes, additional cuts may not strengthen the relaxation, making cut generation worthless. On the other hand, branching avoids the cost of separation routines and immediately partitions the search space. However,

it may result in an exponential number of nodes in the enumeration tree.

To illustrate the impact of cut generation on the branch-and-cut performance, we consider two well-known combinatorial cut classes: subtour elimination constraints for the TSP and cycle inequalities for the Max-Cut problem.

Example 5.1.1. *(Subtour elimination constraint generation for the TSP)*

Given an undirected graph $G = (V, E)$ with a cost vector $\boldsymbol{c} = (c_e)_{e \in E}$ associated with E, the TSP seeks a Hamiltonian cycle (a.k.a. tour) that minimizes the total edge cost. For all edges $e \in E$, we denote by x_e a binary variable such that $x_e = 1$ if edge e occurs in the tour and $x_e = 0$ otherwise. The TSP can be formulated as an integer program as follows:

$$\min \boldsymbol{c}^T \boldsymbol{x} \tag{5.1a}$$

$$\text{s.t.} \sum_{x \in \delta(v)} x_e = 2 \qquad \forall\, v \in V \tag{5.1b}$$

$$\sum_{x \in \delta(S)} x_e \geq 2 \qquad \forall\, \emptyset \neq S \subset V \tag{5.1c}$$

$$x_e \in \{0, 1\} \qquad \forall\, e \in E \tag{5.1d}$$

where $\boldsymbol{x} = (x_e)_{e \in E}$. The objective function (5.1a) represents the total cost of edges selected in the tour. Constraints (5.1b) are *degree constraints* assuring that each vertex in the tour is the end-vertex of precisely two edges. Constraints (5.1c) are *subtour elimination constraints (SECs)*, which guarantee the non-existence of cycles that visit only a proper subset of V. Finally, (5.1d) are *integrality constraints*. Note that this formulation, introduced by Dantzig, Fulkerson, and Johnson [47], is widely used in most branch-and-cut algorithms for the TSP.

To find violated SECs at a node of the enumeration tree, one can use an exact polynomial time algorithm proposed by Crowder and Padberg [69]. The input of the separation algorithm is the optimal solution $\boldsymbol{x}^{\text{LP}}$ of the current LP relaxation. We then construct from $\boldsymbol{x}^{\text{LP}}$ a graph $G_{\boldsymbol{x}^{\text{LP}}} = (V, E_{\boldsymbol{x}^{\text{LP}}})$ where $E_{\boldsymbol{x}^{\text{LP}}} = \{e \in E \mid x_e^{\text{LP}} > 0\}$. For each edge e in $E_{\boldsymbol{x}^{\text{LP}}}$, we set x_e^{LP} as its capacity. Due to the construction of $G_{\boldsymbol{x}^{\text{LP}}}$, the value $\sum_{x \in \delta(S)} x_e$ for $S \subset V$ is precisely the capacity of the cut $(S, V \setminus S)$ in $G_{\boldsymbol{x}^{\text{LP}}}$. Therefore, an SEC violated by $\boldsymbol{x}^{\text{LP}}$ is equivalent to a cut with a capacity smaller than 2 in $G_{\boldsymbol{x}^{\text{LP}}}$. Such a cut can be found by using the Gomory-Hu procedure [51] with $|V| - 1$ maximum flow computations.

To evaluate the impact of SEC generation, we use three different strategies to solve (5.1) with a specific instance, i.e., rat195 from TSPLIB [46]. Firstly, to demonstrate the necessity of SECs, in the *No cut* strategy, we do not generate any cuts. The second one, *Every node*, is a greedy strategy that generates cuts at every search tree node. To illustrate that the impact of cut generation does not only depend on the number of separation routine calls, we use a random strategy called *Sample cut*. This strategy executes separation precisely 100 times - at each tree node, there is a 50% probability of generating SECs, repeating this process until the 100th separation problem is solved. Note that these strategies are only applied to nodes whose LP optimal solution is fractional. At nodes obtaining an integer solution, verifying and generating SECs are mandatory to assure the feasibility of solutions. We use the commercial MILP solver CPLEX12.10 to implement branch-and-cut with SECs and set the CPU time limit to 3600 seconds.

Table 5.1 shows the results of the strategies. *Sample 1* and *Sample 2* are two different runs of the strategy *Sample cut*. Column *"CPU time"* gives the running time in seconds of branch-and-cut, in which the portion of time spent by the separation routine is shown in column *"Sepa time (%)"*. Column *"# Nodes"* reports the number of nodes in the enumeration tree, and column *"# Cuts"* indicates the number of generated SECs. Columns *"# Sepa"* and *"Sepa cuts (%)"* give the number of separation routine executions and the percentage of executions that can obtain violated SECs, respectively.

Strategy	**CPU time**	**Sepa time (%)**	**# Nodes**	**# Cuts**	**# Sepa**	**Sepa cuts (%)**
No cut	3601.8*	0.0%	1506514	0	0	-
Every node	1365.7	98.2%	4105	1116	2992	4.5%
Sample 1	65.5	73.7%	3834	359	100	21.0%
Sample 2	114.5	34.5%	10543	727	100	43.0%

Table 5.1. The results of the SEC generation strategies on the instance rat195. The asterisk in the "*CPU time*" column indicates strategies that fail to solve the TSP within the time limit.

Table 5.1 shows that the SEC generation strategies may significantly affect the algorithm performance. Obviously, generating SECs is crucial, as branch-and-cut cannot solve the instance to optimality without it under the given CPU time budget. In addition, adding SECs substantially reduced the enumeration tree size. However, solving SEC separation problems might take a major portion of computing time, and only a few separation executions obtained violated SECs. For example, with the strategy generating SECs at every tree node, branch-and-cut spent 98% of the CPU time to execute separation routines, but only 4.5% of executions yielded violated SECs. Table 5.1 also indicates that the effectiveness of the strategies relies not only on the number of solved separation problems but also on specific nodes where violated SECs are generated. Indeed, although the number of times the separation problem is solved is the same, the difference in nodes generating SECs makes the strategy *Sample 1* outperform *Sample 2*.

Example 5.1.2. *(Cycle inequality generation for the Max-Cut problem)*

Given an undirected graph $G = (V, E)$ and each edge e is associated with a weight $w_e \in \mathbb{R}_+$, a cut in G is a partition of V into two disjoint subsets $S \subset V$ and $\overline{S} = V \setminus S$. The weight of a cut $(S, \overline{S})$ is the sum of weights of edges with one end-vertex in S and the other in $\overline{S}$. The Max-Cut problem is to find a cut $(S, \overline{S})$ in G with maximum weight.

A commonly used IP formulation for the Max-Cut problem based on the semi-metric polytope [70] is the following:

$$\max \ \boldsymbol{w}^T \boldsymbol{x} \tag{5.2a}$$

$$\text{s.t.} \sum_{e \in F} x_e - \sum_{e \in C \setminus F} x_e \leq |F| - 1 \qquad \forall C \in \mathbf{C}, F \subset C \text{ with } |F| \text{ odd.} \tag{5.2b}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E \tag{5.2c}$$

where $\mathbf{C}$ is the set of chordless cycles in G. Constraints (5.2b) are *cycle inequalities* proposed by Barahona and Mahjoub [70], which represents the fact that any cut intersects a cycle in an

even number of edges. Similar to SECs of the TSP, cycle inequalities will be generated as cuts in branch-and-cut due to their exponential cardinality.

The separation problem for cycle inequalities can be solved by the following exact polynomial algorithm, which is introduced by Barahona and Mahjoub [70]. Given an optimal solution $\boldsymbol{x}^{\text{LP}}$ of the LP relaxation, we construct a graph $G_{\boldsymbol{x}^{\text{LP}}}$ as follows: for each vertex i in G, we create two vertices i' and i'' in $G_{\boldsymbol{x}^{\text{LP}}}$; for each edge ij in G, we add to $G_{\boldsymbol{x}^{\text{LP}}}$ edges $i'j'$ and $i''j''$ with weight x_{ij}^{LP}, and edges $i'j''$ and $i''j'$ with weight $1 - x_{ij}^{\text{LP}}$. By this setting, a cycle inequality violated by $\boldsymbol{x}^{\text{LP}}$ is the shortest path with length less than 1 from i' to i'' in $G_{\boldsymbol{x}^{\text{LP}}}$. Hence, the algorithm consists of $|V|$ shortest path computations in $G_{x^{\text{LP}}}$.

Strategy	**CPU time**	**Sepa time (%)**	**# Nodes**	**# Cuts**	**# Sepa**	**Sepa cuts (%)**
No cut	3600.0*	0.0%	302913	0	0	-
Every node	382.1	69.8%	553	23461	1246	67.1%
Sample 1	174.5	43.5%	650	10958	300	84.7%
Sample 2	493.1	35.4%	636	13379	300	89.0%

Table 5.2. The results of the cycle inequality generation strategies on the instance pm1s_100.5. The asterisk in the "*CPU time*" column indicates strategies that fail to solve the problem to optimality within the time limit.

Similar to the previous example, we solve the Max-Cut problem on instance pm1s_100.5 generated using rudy [71]. We also evaluate the performance of the strategies: *No cut*, *Every node*, and *Sample*. In the strategy *Sample*, we increase the number of times separation problems are solved to 300.

Table 5.2 again confirms that generating cuts is indispensable for solving the Max-Cut problem to optimality within the time limit. The results of the *Every node* and *Sample* strategies also demonstrate that both the number of solved separation problems and the position of nodes where cuts are generated strongly impact the algorithm's performance. One difference between SEC generation in the TSP and cycle inequality generation in the Max-Cut problem is that most separation problems for cycle inequalities yield valid cuts. In contrast, only a small fraction of SEC separation problems in the TSP yield violated constraints.

Motivated by these empirical observations, we study the cut generation problem stated as follows. Consider a combinatorial optimization problem $\mathcal{P}$ defined on a graph $G = (V, E)$ and a cut class $\mathcal{C}$ associated with $\mathcal{P}$. A cut generation strategy $\pi_{(\mathcal{P},\mathcal{C})}$ for $\mathcal{C}$ in branch-and-cut decides whether to generate $\mathcal{C}$-type cuts or to branch at each node of the enumeration tree. The cut generation problem is to learn a cut generation strategy $\pi_{(\mathcal{P},\mathcal{C})}$ that obtains the best average performance τ on a given set of problem instances $\mathcal{I}_{\mathcal{P}}$, i.e.,

$$\pi_{(\mathcal{P},\mathcal{C})} \in \arg\min_{\pi \in \Pi} \mathbb{E}_{p \in \mathcal{I}_{\mathcal{P}}}[\tau(p, \pi)]$$

where $\tau(p, \pi)$ is the running time of branch-and-cut for solving instance p with cut generation policy π. Note that although we restrict our work to graph problems, this problem type still can cover a wide range of real-world problems.

Throughout this chapter, to avoid any confusion, we only consider one class of combinatorial cuts in a branch-and-cut algorithm for a combinatorial optimization problem. Unless otherwise specified, cuts refer to those from the particular cut class under consideration.

5.2 Literature review

Most approaches to cut generation in the literature exist in heuristic forms. Padberg and Rinaldi, in their research on branch-and-cut for large-scale TSP [72], empirically discovered the tailing-off phenomenon of cuts [72, Section 4.3], which shows the cut generator's inability to produce cuts that can assist the optimal LP solution to escape the corner of the polytope where it is "trapped". To deal with the tailing-off, the authors proposed stopping the generation of cuts if the objective value of the relaxed LP does not improve sufficiently within a given window and switching to branching. Another approach to control cut generation introduced by Balas et al. [73] is generating cuts at every k nodes of the enumeration tree. The number k, named "skip factor" in [73], determines the frequency of generating cuts. It can be chosen either as a fixed constant or as an adaptive value varying throughout the enumeration tree. Another commonly used strategy is the so-called cut-and-branch, which only generates cuts at the root node of the enumeration tree. Overall, despite its importance, the question of the branching versus cutting decision has yet to receive the attention it deserves.

In contrast with cut generation, *cut selection*, a closely related problem, has been studied extensively in the literature. While cut generation decides whether to launch separation processes to generate cuts, cut selection requires selecting cuts from a candidate set obtained by solving separation problems. Cut selection is usually considered for general-purpose cuts whose separation procedure is computationally cheap and provides many cuts. Due to its definition, cut selection can be viewed as a ranking problem where cuts are sorted and chosen based on some criteria. This point of view opened up many different approaches based on many measurements of the cut quality. Among the most popular scores are efficacy [73], objective parallelism [74], and integral support [75], to name a few. Another research line on cut selection is to use machine learning to learn the ranking of cuts. Most works of this approach fall into two categories: supervised learning and RL. In the former, cuts are scored (or labeled) by an expert, and a cut ranking function (usually a neural network) is trained to be able to choose the best ones [67]. For the latter, one can formulate the problem of sequentially selecting cuts as a Markov decision process. An agent can then be trained to either directly optimize the objective value (RL) [24] or mimic a look-ahead expert (imitation learning) [68].

In recent years, using machine learning to enhance fundamental decisions in branch-and-bound is an active research domain; we refer to [76] for a summary of this line of work. Specific examples contain learning to branch [21, 22, 64], learning to select nodes [23], and learning to run primal heuristics [65, 66]. Table 5.3 briefly presents several approaches to using machine learning for branch-and-cut inner strategies. Similar to cut selection, these problems can be reformulated as ranking [21, 22, 67], regression [64], or classification problems [65], and can then be treated correspondingly. Most of these reformulations are possible due to the existence of an expensive expert (for example, the strong branching expert for variable selection), which can be used to calculate the score, label the instances, or act as an agent to be mimicked.

Decision problem	Work	Learning paradigm	Requirements	Method description
Variable selection	Khalil et al. 2016 [21]	Imitation learning	An expert; train and test the model on the same instance.	This work considers variable selection as a *ranking problem* where the strong branching expert determines the label of candidates. Each variable candidate is represented as a vector whose elements contain statistical features of this variable in the enumeration tree. The model is trained with the expert's guide at the beginning of the branch-and-bound process and then used to the late stage of the solving process.
	Alvarez et al. 2017 [64]	Imitation learning	An expert; training and testing instances must have the same size.	This work considers variable selection as a *regression problem* that directly predicts the strong branching score of candidates. Each variable candidate is represented as a vector whose elements contain statistical features of this variable in the enumeration tree. The training data are collected by running the expert on a set of fixed-size instances. The trained model is then used on same-size instances.
	Gasse et al. 2019 [22]	Imitation learning	An expert.	This work considers variable selection as a classification problem where the label of candidates is assigned by the strong branching expert. Each tree node is represented as a bipartite graph where one side represents nodes, and another represents constraints. The training data are collected by running the expert on a set of fixed-sized instances. The trained model can be used for various-size instances.
	Etheve et al. 2020 [77]	Reinforcement learning	The node selection is restricted to the depth-first search; training and testing instances must have the same size.	This work presents variable selection as an MDP. The reward function is defined by the number of nodes in the tree generated when solving the instance to optimality. Each state is represented as a vector concatenated from dynamic and statistical features. The agent is trained by the DQN algorithm.
Node selection	He et al. 2014 [23]	Imitation learning	An expert.	This work constructs an oracle that expands nodes whose feasible set contains the optimal solution. Then, two policies are learned to mimic the oracle: a node selection policy determining the highest-priority node and a node pruning policy deciding whether to prune or expand this node. States are represented as vectors that include node, branching, and tree features.
	Song et al. 2018 [78]	Imitation learning	An expert.	This work constructs a retrospective oracle that generates feedback to train policies by querying the environment on rolled-out search traces to find the shortest path from the root node to a terminal state. This approach can be scaled up by iteratively learning to solve increasingly larger instances.
	Labassi et al. 2022 [79]	Imitation learning	An expert.	This work constructs a classifier function to compare two open nodes in the enumeration tree. Node labels are assigned by using the diving oracle as in He et al. [23]. Each node is represented as a bipartite graph, as in Gasse et al [22].
Cut selection	Tang et al. 2020 [24]	Reinforcement learning		This work considers cut selection in the cutting plane methods as an MDP where the reward function is defined based on objective improvement. An agent is then trained by evolution strategies. The trained model can be applied to instances with various sizes.

Paulus et al. 2022 [68]	Imitation learning	An expert.	This work constructs a strong-branching-like expert for cut selection that computes the look-ahead score of a cut, which is the objective improvement obtained by adding this cut. Then, cut selection is reformulated as a regression problem to predict the look-ahead score of a cut. Each state is represented by a tripartite graph whose nodes represent variables, constraints, and additional cuts.
Huang et al. 2022 [67]	Supervised learning	A labeled dataset.	This work focuses on the cut selection task at the root node of branch-and-cut. This task is considered as a ranking problem where cuts are assigned labels by added and solved beforehand. Each cut is represented as a vector whose components contain dynamic and statistical features.
Wang et al. 2023 [80]	Reinforcement learning		The work proposes a hierarchical sequence model consisting of a two-level model: a higher-lever model to learn the number of cuts that should be selected and a lower-level model to decide which cuts should be added and what order of chosen cuts. The key idea is to represent states as sequences of candidate cuts, which can encode the underlying order information and the interactions among cuts. The model is trained by reinforcement learning with a hierarchical policy gradient.

Table 5.3. Machine learning for branch-and-cut inner strategies

5.3 General framework for cut generation

In this section, we propose a general framework based on machine learning to address cut generation and explain the key ideas behind the proposed method. We begin by presenting an MDP formulation associated with cut generation in Section 5.3.1. Then, in Section 5.3.2, we demonstrate why we do not use imitation learning to train the agent as in most previous works. Finally, we present our framework integrating both supervised and reinforcement learning for learning cut generation policies in Section 5.3.3.

5.3.1 Markov Decision Process formulation

Notice that the sequential decisions of cut generation can be assimilated to an MDP. Considering the solver as the environment and a cut generation policy as an agent, we define the state space, action space, and transition of the MDP as follows:

State space $\mathcal{S}$

At time step t, a state $s_t \in \mathcal{S}$ is the entire current enumeration tree, which comprises LP relaxations at nodes, branching decisions, the lower and upper bounds, added cuts, the incumbent solution, the currently focused node with an optimal solution to the corresponding LP relaxation, and other solver statistics. The terminal state is achieved when the instance is solved to proven optimality.

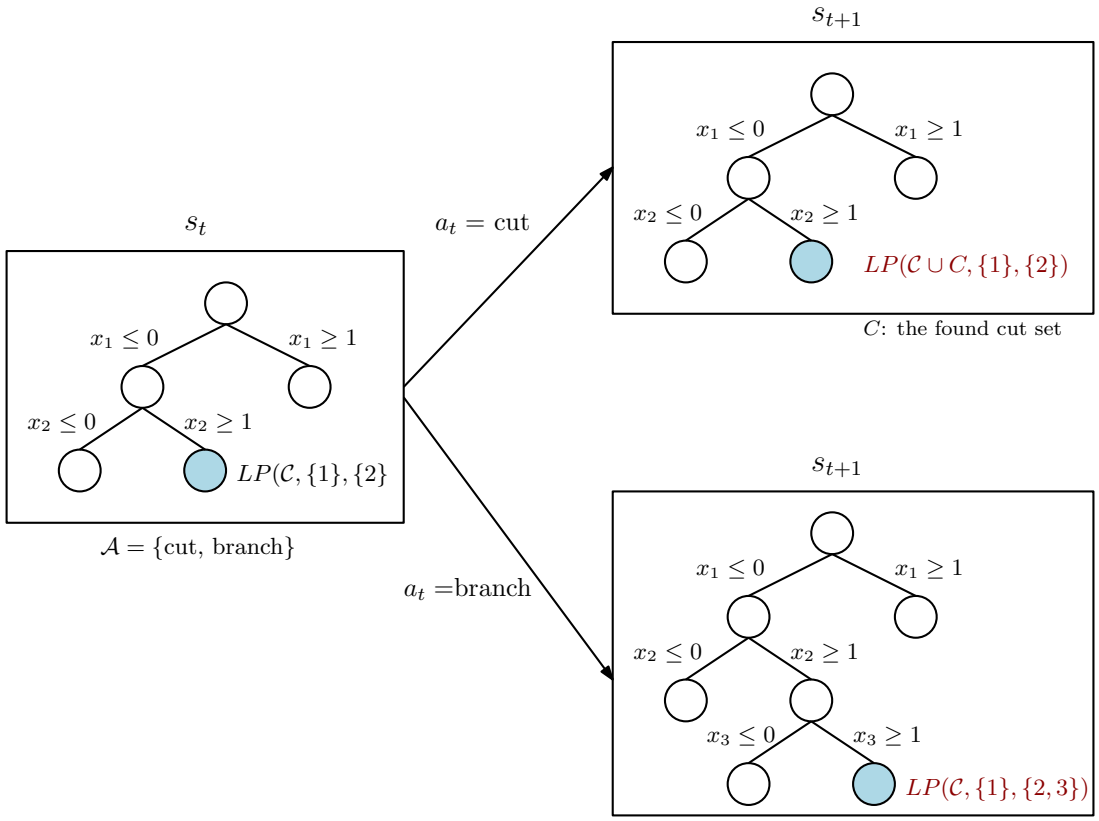


Figure 5.1. A transition in the MDP of cut generation.

Action space $\mathcal{A}$

At a non-terminal state s_t, the agent chooses action a_t from two options: *generating cuts* or *branching*.

Transition

After selecting an action a_t, the new state s_{t+1} is determined as follows. If a_t is to branch, the solver selects a branching variable to create two child nodes, uses the node selection policy to pick an active node (i.e., that has not been pruned nor branched on), and solves the corresponding LP relaxation to get an optimal solution. Otherwise, if a_t is to generate cuts, the solver launches the separation routine to find cuts, adds them to the formulation, and solves the LP relaxation again with additional cuts to obtain an optimal solution. If no cut is found, the next state s_{t+1} is determined in the same way as when performing the branching action. Figure 5.1 illustrates a transition of this MDP.

5.3.2 Why don't we learn a cut generation policy by imitation learning?

As presented in Section 5.2, most of the previous approaches to learn branch-and-cut policies are possible due to the existence of an expert, which can be used to calculate the score, label the instances, or act as an agent to be mimicked. The natural question is that:

Could we design such an expert for cut generation and replace it with a learned heuristic?

The main intuition of expert strategies for branch-and-cut decisions like strong branching for variable selection [81] or lookahead rules for cut selection [68], is to try beforehand decisions and select the one with maximum benefit. A common decision impact measure is the bound improvement obtained from the resulting LP relaxations.

We follow similar intuition to construct an expert for cut generation. At each node, the expert attempts both actions of generating cuts and branching in advance. It then computes action scores in terms of the objective improvement and selects the action with a higher score. Formally, given the current cut pool $\mathcal{C}$, the LP relaxation corresponding to the considered node $LP(\mathcal{C}, F_0, F_1)$, a set of cuts C found by the separation routine and a branching candidate i, the scores s of actions are computed as follows:

$$\begin{aligned} s_{\text{cuts}} &= \left| z_{LP(\mathcal{C}\cup C, F_0, F_1)} - z_{LP(\mathcal{C}, F_0, F_1)} \right| \\ s_{\text{branch}} &= \max\left(\left| z_{LP(\mathcal{C}, F_0\cup\{i\}, F_1)} - z_{LP(\mathcal{C}, F_0, F_1)} \right|, \left| z_{LP(\mathcal{C}, F_0, F_1\cup\{i\})} - z_{LP(\mathcal{C}, F_0, F_1)} \right| \right) \end{aligned}$$

where $z_{LP(.)}$ is the objective value of the relaxation $LP(.)$. In the case the cut set C is empty, the score of generating cuts is set to $-\infty$. If an action can yield an integer feasible solution or fathom a node, its score is $+\infty$. If the two actions have equal scores, the expert will prioritize generating cuts.

To evaluate the expert policy, we use it to control the generation of two classes of cuts: subtour elimination constraints for the TSP and cycle inequalities for the Max-Cut problem. Experiments are conducted on 100 random TSP instances (complete graphs of 200 vertices) generated by Johnson and McGeoch's generator [82] and 100 random Max-Cut instances (10%−dense graphs of 100 vertices) generated by the rudy generator [83]. To focus on decision quality, each instance is solved twice: the first run uses the expert to record decisions, while the second run replays those decisions to exclude decision time. Then, the results of the second runs will be used to compare with baselines.

We compare the expert performance with the fixed and automatic strategies proposed in [73], which generate cuts for every k nodes. For the fixed strategies, we use $k = 1$ (`FS-1`) as the default strategy and $k = 8$ (`FS-8`), which gave the best results in [73] and is one of the best skip factors in our own experiments. For the automatic strategy (`AS`) where the skip factor is chosen based on the instance to be solved, k is computed as follows [73]:

$$k = \min\left\{ \text{KMAX}, \left\lceil \frac{f}{cd \log_{10} p} \right\rceil \right\} \tag{5.3}$$

where f is the number of cuts generated at the root node, d is the average distance cutoff of these cuts (a distance cutoff of a cut is the Euclidean distance between the optimal solution and the cut), p is the number of variables in the formulation, and KMAX, c are constants. Based on the cut properties and our experiments, we set (KMAX, c) to $(32, 100)$ for SECs and $(32, 1000)$ for cycle inequalities.

Table 5.4 shows the results of the expert and three baselines on the TSP and Max-Cut instances. For each instance type, we report the average CPU time in seconds (column *"CPU Time"*), the average number of nodes in the enumeration tree (column *"# Nodes"*), and the average number of generated cuts (column *"# Cuts"*).

Cut type	Strategy	CPU Time	# Nodes	# Cuts
SECs	FS-1	109.9	**2769.0**	506.9
	FS-8	56.8	3090.1	493.8
	AS	48.1	3521.1	439.3
	Expert	**44.4**	6172.5	**401.8**
Cycle inequalities	FS-1	275.4	**297.4**	14403.4
	FS-8	105.9	447.8	9107.7
	AS	**87.1**	385.9	9959.9
	Expert	123.5	400.2	**8106.8**

Table 5.4. The numerical results of the expert policy

As shown in Table 5.4, the expert policy slightly outperforms the heuristics for subtour elimination constraint generation but is significantly worse than the best baseline for cycle inequality generation. The deeper analysis further points out that the expert is expensive, requiring 429.4 average seconds for SEC decisions and 3083.5 seconds for cycle inequalities. Therefore, the cost of the expert is not commensurate with the quality of the decisions it makes. From this experiment, we decide to use reinforcement learning, a natural approach, to train the agent.

5.3.3 Hybrid framework for cut generation

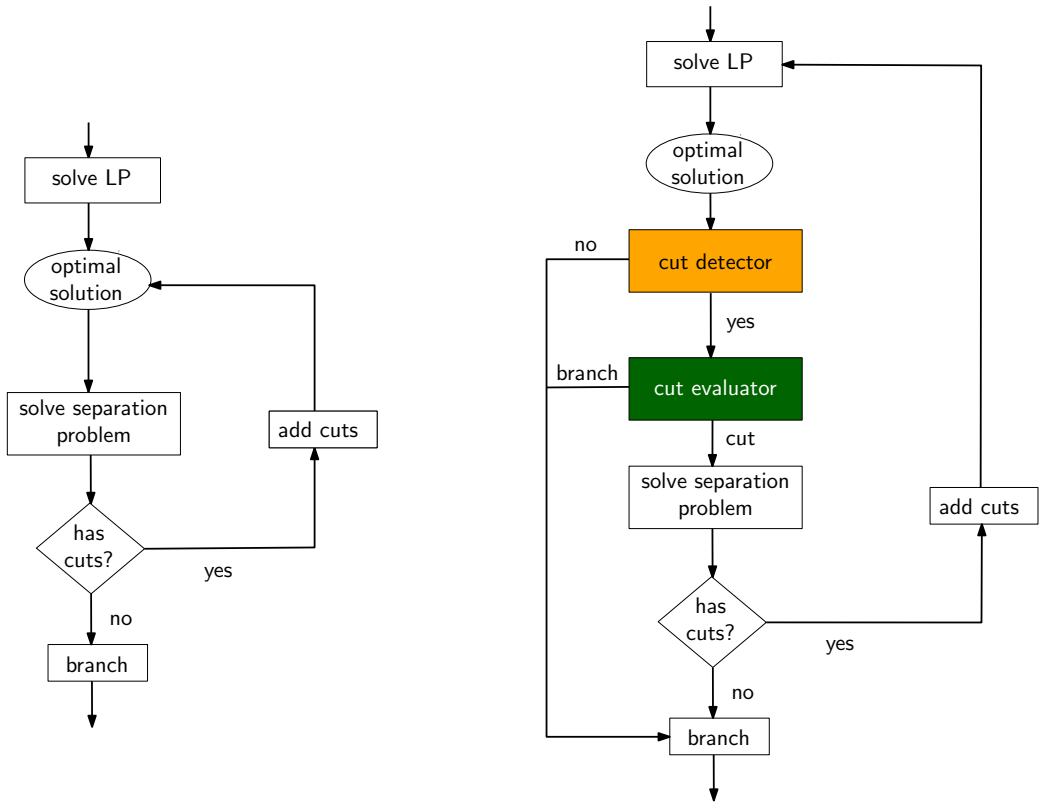


Figure 5.2. The flowchart of exploring a tree node without (left) and with (right) our framework

We now present an overview of our framework for tackling the cut generation problem in

branch-and-cut. The inner components will be fully described in Section 5.4 and Section 5.5.

We first observe that a principal requirement of an efficient cut generation is the existence of cuts violated by the LP optimal solution. This requirement aims to avoid solving redundant separation problems. For general-purpose cuts, this requirement is usually omitted since separation routines are computationally cheap and guarantee to find a cut. In contrast, separation problems for combinatorial cuts are more expensive and may fail to produce any cuts, as demonstrated in the experiment on SECs in Section 5.1.

Based on this observation, our framework consists of *two separate components*:

- a *cut detector*, a GNN that predicts the cut existence,
- a *cut evaluator*, an RL agent that decides whether to generate cuts.

The cut detector serves as a coarse filter before the cut evaluator makes decisions. The usage of two components allows us to use lightweight models yet still guarantee accuracy. Furthermore, while the benefit evaluation of generating cuts requires both local and global information about the search tree, detecting the existence of cuts only needs an optimal solution of LP relaxation. Consequently, when operating on trees containing thousands of nodes, we can save considerable time by reducing information extraction. Note that the cut detector is optional, depending on the cut properties. It is crucial for cut types where the corresponding separation procedure infrequently succeeds, like SECs for the TSP. However, it is unnecessary for cut types where separation executions always obtain cuts, like cycle inequalities for the Max-Cut problem.

With this architecture, our framework uses a mixture of supervised and reinforcement learning. Specifically, the cut detector is a classifier trained by supervised learning, and the cut evaluator is an agent trained by reinforcement learning.

Figure 5.2 illustrates the flowchart of our framework at an enumeration tree node. After obtaining an optimal solution to the LP relaxation at the node, the cut detector predicts whether the solution violates any cuts. If it predicts no cut violation, we proceed directly to the branching step. Otherwise, the cut evaluator assesses the effectiveness of additional cuts and selects the next action to take. Note that this flow is only applied when the LP optimal solution is fractional. For abbreviation, we call a fractional solution an LP optimal solution that is fractional.

5.4 Cut detector

In this section, we present the first component of our framework, *the cut detector*. Given an optimal solution $\boldsymbol{x}^{\mathrm{LP}}$ to the LP relaxation corresponding to a node in the enumeration tree, the cut detector aims to predict whether there exists any cut violated by $\boldsymbol{x}^{\mathrm{LP}}$. This component can be viewed as a *binary classifier* that takes a fractional solution $\boldsymbol{x}^{\mathrm{LP}}$ as inputs and returns:

$$y = \begin{cases} 1 & \text{if there exists any cut violated by } \boldsymbol{x}^{\mathrm{LP}}, \\ 0 & \text{otherwise.} \end{cases}$$

Therefore, we can train the cut detector in a supervised manner. Given a training set $\mathcal{D} = \{(\boldsymbol{x}_i^{\mathrm{LP}}, y_i)\}_{i=1}^{N}$ where $\boldsymbol{x}_i^{\mathrm{LP}}$ is a fractional solution, and y_i is its label, our goal is to train a binary classifier $\mathbf{f}_{\Theta_D}(\boldsymbol{x}^{\mathrm{LP}})$, which can predict the label of each fractional solution $\boldsymbol{x}^{\mathrm{LP}}$.

5.4.1 Constructing training data

To collect training samples, we use a random cut generation strategy (i.e., generate cuts with a probability of 0.5 at each node) on training instances, which are generated randomly. For each problem instance, we run branch-and-cut with the random cut generation policy multiple times to collect LP optimal solutions. Labels of these solutions are assigned by executing the separation routine. Since the cut generation policy is stochastic, we can explore diverse enumeration trees and obtain varied training samples. Although collecting samples requires multiple branch-and-cut executions, this process is offline, so the training cost is acceptable.

5.4.2 Cut detector architecture

Recall that each fractional solution $\boldsymbol{x}^{\text{LP}}$ is a vector whose components correspond to decision variables. As a result, its dimension depends on the instance graph size, which can grow extremely large. Furthermore, this raw representation can not encode relationships between variables based on the underlying graph structure (e.g., x_i^{LP} and x_j^{LP} may represent decision variables for two adjacent edges in the instance graph), which is a key context for the separation routine.

To address these issues, rather than directly use the raw high-dimensional vector $\boldsymbol{x}^{\text{LP}}$, we propose representing $\boldsymbol{x}^{\text{LP}}$ by the graph constructed from $\boldsymbol{x}^{\text{LP}}$ to solve the separation problem. We call this graph the *separation-purpose graph*. We then use a GNN to embed this graph into a vector. By operating on the separation-purpose graph, the GNN incorporates the variable relationships and graph context needed for the separation routine by operating on the separation-purpose graph. The resulting embedding provides a descriptive representation of $\boldsymbol{x}^{\text{LP}}$, which is independent of instance size and permutation invariant.

We parameterize the cut detector as follows. Given a fractional solution $\boldsymbol{x}^{\text{LP}}$, we represent it as its separation-purpose graph $G_{\boldsymbol{x}^{\text{LP}}} = (V_{\boldsymbol{x}^{\text{LP}}}, E_{\boldsymbol{x}^{\text{LP}}})$ with node features $\mathbf{V} \in \mathbb{R}^{|V_{\boldsymbol{x}^{\text{LP}}}| \times d_n}$ and edge features $\mathbf{E} \in \mathbb{R}^{|E_{\boldsymbol{x}^{\text{LP}}}| \times d_e}$, where $d_n, d_e \in \mathbb{R}^+$ are the dimension of node and edge features, respectively. For each node $i \in V_{\boldsymbol{x}^{\text{LP}}}$, we embed its features $\mathbf{v}_i \in \mathbb{R}^{d_n}$ to a h-dimensional vector by an MLP:

$$h_i^{(0)} = \Theta_D^{(0)} \mathbf{v}_i + b_D^{(0)}$$

where $\Theta_D^{(0)} \in \mathbb{R}^{h \times d_n}$ and $b_D^{(0)} \in \mathbb{R}^h$. To update the node embedding, we use two MP layers in the form:

$$h_i^{(l)} = \text{MERGE}_{\Theta_D^{(l,1)}} \left(h_i^{(l-1)}, \text{AGGR}_{\Theta_D^{(l,2)}} \left(\left\{ \left(h_j^{(l-1)}, \mathbf{e}_{i,j} \right) \right\} | j \in \mathcal{N}(i) \right) \right)$$

where $\text{MERGE}_{\Theta_D^{(l,1)}}$ and $\text{AGGR}_{\Theta_D^{(l,2)}}$ are the merge and aggregate functions, $\mathbf{e}_{i,j}$ is the feature vector of edge $(i,j) \in E_{\boldsymbol{x}^{\text{LP}}}$, and $\mathcal{N}$ is the set of i's neighbors in $G_{\boldsymbol{x}^{\text{LP}}}$. To yield a graph embedding, we apply a permutation invariant pooling layer to aggregate all node embeddings. This graph representation is input into an MLP with a softmax activation function to compute the probability of the solution's labels, i.e., $P(y = 0 | \boldsymbol{x}^{\text{LP}})$ and $P(y = 1 | \boldsymbol{x}^{\text{LP}})$. Figure 5.3 illustrates the cut detector architecture.

The node and edge features of the separation graph are chosen depending on the separation routine, which varies between combinatorial cut types. We provide examples of feature selection for several cut types in the experiment section.

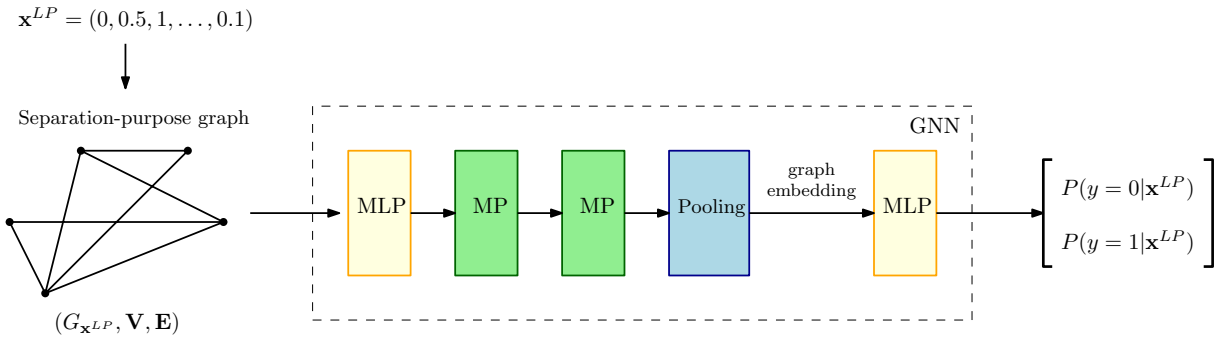


Figure 5.3. The cut detector architecture

5.4.2.1 Training the cut detector

Given a training set $\mathcal{D} = \{(\boldsymbol{x}_i^{\text{LP}}, y_i)\}_{i=1}^N$, we optimize the cut detector parameters Θ_D to minimize the loss function involving the cross-entropy and regularization losses, i.e.,

$$L(\Theta_D) = L_E(\Theta_D) + \eta\Omega(\Theta_D)$$

where η is a hyper-parameter for regularization penalty, $L_E(\Theta_D)$ is computed by

$$L_E(\Theta_D) = -\sum_{i=1}^{N} \Big(y_i \cdot \log P_{\Theta_D}(y_i = 1|\boldsymbol{x}_i^{\text{LP}}) + (1 - y_i) \cdot \log(1 - P_{\Theta_D}(y_i = 0|\boldsymbol{x}_i^{\text{LP}})) \Big)$$

and $\Omega(\Theta_D)$ is calculated by

$$\Omega(\Theta_D) = \|\Theta_D\|_2^2 .$$

The use of the regularization loss aims to avoid the overfitting of the model.

5.5 Cut evaluator

We now present the key component of our framework - the *cut evaluator*. It is a deep RL agent that learns an optimal policy for cut generation. To train the cut evaluator by RL, we need to address two main challenges:

1. ***How do we define the reward function to align with the ultimate goal of minimizing runtime?***

   Central to RL is the reward function, which specifies the objective to be optimized. In the cut generation problem, our goal is to minimize branch-and-cut runtime. Thus, a naive approach is to use the runtime directly to define rewards - specifically, defining the reward of an action as the negative of the time spent performing that action. However, our experiments show that the agent struggles to learn good policies with this reward function for two reasons. First, generating cuts has long-term benefits that are usually only observable at the endpoints of trajectories. Nevertheless, trajectories of cut generation are generally very long, causing the agent to fall into suboptimal policies that only branch, as the runtime of branching is always shorter than that of generating cuts. Second, this reward function depends on hardware and is thus non-deterministic. It means the same action taken on different hardware or at different times may result in different rewards,

making it difficult for the agent to learn and generalize. To tackle these issues, we propose two approaches: 1) using an alternative reward function based on the IP relative gap and 2) retaining the time-based reward function but using additional hypotheses to simplify the MDP. The two directions allow our framework to adapt to various properties of cut types.

2. ***How do we represent states?***

   As defined in Section 5.3.1, a state contains the entire search tree, a complex structure that can not be fully represented. Thus, we can only represent states by a subset of the search tree. In the following sections, we present two state representations corresponding to the two proposed reward functions.

5.5.1 The gap-based reward function

5.5.1.1 Reward function definition

Our first reward function definition is based on the IP relative gap, which is the ratio between the upper and lower bound absolute difference and the current best objective value. Branch-and-cut algorithms terminate when the IP relative gap reaches 0, so the faster the gap decreases, the faster the instance can be solved. Given this, we define the reward of taking action a_t in state s_t as the resulting change in the IP relative gap, i.e.,

$$r_t = r(s_t, a_t) = \mu_t - \mu_{t+1}\,. \tag{5.4}$$

where μ_t is the IP relative gap at state s_t.

A key issue with this reward definition is its sparsity - most rewards are 0. Consequently, the agent rarely receives feedback, making learning difficult. To address the sparsity, we use shaping rewards to provide appropriate hints to help the agent learn more efficiently. Specifically, we introduce three additional rewards:

- a penalty for each extra iteration, encouraging faster solving;
- a penalty for each time solving a redundant separation problem, discouraging unnecessary computations;
- a bonus for each cut found, incentivizing productive cut generation.

Together, these shaped rewards guide the agent towards balanced policies that generate cuts efficiently. The iteration penalty promotes speed, while separation and cut rewards tune the trade-off between the computational expense of separation versus the benefit of additional cuts.

5.5.1.2 State representation

Due to the enumeration tree complexity, we represent a state s_t as a collection of the following elements:

- *An optimal solution $\boldsymbol{x}_t^{LP}$ to the LP relaxation of the considered node.* It is used to provide information about the separation problem for the agent. We represent this solution by its separation-purpose graph as in the cut detector.

- *The problem instance.* Since we focus on graph-structured combinatorial optimization, the problem instance is a graph with features associated with nodes or edges. We also add information about the decision variables at the currently considered node (values in the optimal LP solution, lower and upper bounds) in the node and edge features to encode the context of the considered node in the search tree.

- *Features of the search tree.* To enrich the information about the search tree in the state representation, we design 11 tree features based on our experimental observations and inspired by hand-crafted input features for branching variable selection proposed in [84]. The features are shown in Table 5.5. The top four features correspond to the incumbent existence, the IP relative gap, and the portions of processed and unprocessed nodes, which help to capture the state of the search tree. The remaining features are extracted at the considered node to describe its context through depth, objective value, optimal solution, and fixed variables. Each feature is normalized to the range $[0, 1]$.

Feature group	**Feature**	**Description**	**Ref.**
Tree (4)	has_incumbent	1 if an integer feasible solution is found and 0 otherwise	
	IP_rel_gap	(upper bound - lower bound) / upper bound	[84]
	processed_nodes	the proportion of processed nodes in the tree	[84]
	unprocessed_nodes	the proportion of unprocessed nodes in the tree	[84]
Node (7)	node_depth	max(1, the node depth / n)	[84]
	obj_quality	objective value / upper bound	
	vars_1	the number of variables equal to 1 in the solution / N	
	fixed_vars	the number of fixed variables / n	
	unfixed_vars	the number of unfixed variables / n	
	vars_fixed_1	the number of variables fixed to 1 / N	
	vars_fixed_0	the number of variables fixed to 0 / n	

Table 5.5. The tree features for the gap-based reward function; n is the number of decision variables and N is the maximum number of non-zero values in an optimal solution.

5.5.1.3 Discussion

The gap-based reward function with reward shaping is hardware-independent and can provide valuable feedback for the cut evaluator to learn. It is well-suited for cuts that significantly strengthen LP relaxations but are rare and expensive to generate, e.g., SECs for the TSP.

However, this reward function relies heavily on properly tuned hyperparameters for the additional shaping rewards. Finding reasonable settings requires extensive tuning that hinders the framework's scalability. Given these generalization limitations, in the next section, we propose an alternative time-based reward definition to improve the framework's flexibility.

5.5.2 The time-based reward function

5.5.2.1 Simplified MDP and the time-based reward function

As claimed above, the agent cannot learn useful policies when directly using the time-based reward function on the original MDP. In this subsection, we propose a way to simplify the MDP by using additional hypotheses and restrictions to help the agent learn.

Given a search tree obtained by solving the instance to optimality, we define a branch as the unique path from a node to one of its descendant leaf nodes in the tree. Recall that a leaf node is a fathomed node that meets one of three conditions: i) its corresponding LP relaxation is infeasible, ii) an integer solution is obtained, and iii) its LP objective value is worse than the current best one. Then, the search tree can be partitioned into a set of disjoint branches, as illustrated in Figure 5.4.

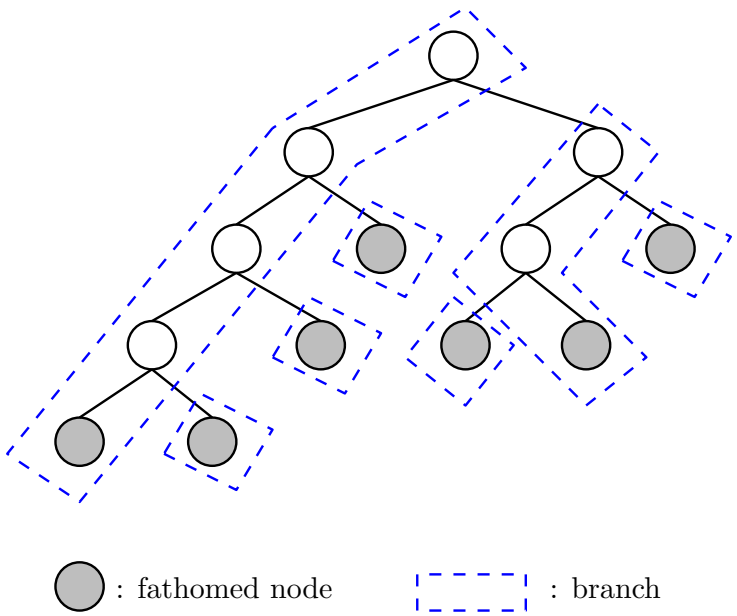


Figure 5.4. An illustration of partitioning the tree to a set of branches

Observe that if the node selection strategy is the depth-first search, exploring the enumeration tree can be viewed as repeatedly exploring branches to reach leaf nodes. From this viewpoint, our idea is to construct a greedy cut generation policy to rapidly explore a branch and yield a good integer solution. As a result, instead of working on the entire enumeration tree, we only need to focus on less complicated branches. Furthermore, a good solution at the early stage of branch-and-cut allows us to prune non-promising nodes before they are expanded, reducing the tree size. Thus, we concentrate on the branch containing the root node, as it is naturally considered more important than other branches.

Based on these intuitions, we simplify the MDP of cut generation as follows. Starting from the root node, the agent iteratively chooses between cut generation and branching, aiming to reach a leaf node. The reward for reaching a leaf node is the negative of the relative IP gap at that node. Each action incurs a cost equal to its runtime. The final reward is the leaf node reward subtracted by the total running time of all performed actions. The agent's ultimate goal is to discover a cut-generation policy that maximizes this final reward.

Compared to the original MDP, the simplified version changes the terminal state definition, state representation, and branch-and-cut node selection strategy restriction. Table 5.6 details the differences between the two versions.

Simplified MDP	Original MDP
A state s_t is the currently considered node	A state s_t is the entire current enumeration tree
The terminal state is achieved when a leaf node is reached	The terminal state is achieved when the instance is solved to optimality
The node selection strategy is restricted to depth-first search	The node selection strategy is not restricted

Table 5.6. The differences between the simplified and original cut generation MDPs.

With the simplified settings, the average length of trajectories is reduced significantly, allowing us to use the running time as rewards. In particular, we define the reward of action a_t at state s_t as follows:

$$r_t = r(s_t, a_t) = \begin{cases} -\mu_t, & \text{if } s_t \text{ is the terminal state} \\ -T(a_t), & \text{otherwise,} \end{cases}$$

where μ_t is the IP relative gap at state s_t and $T(a_t)$ is the runtime of action a_t. If a leaf node is reached without finding a feasible solution, the terminal state is set to $-\infty$.

5.5.2.2 State representation

Since a state in the simplified MDP is the current node instead of the entire search tree, we only use two components to represent a state:

- *An optimal solution $\boldsymbol{x}^{LP}$ to the corresponding LP relaxation.* We still use the information of the optimal solution to represent the state as in the original MDP. However, instead of representing the solution by its separation-purpose graph, which depends on the type of cuts, we represent the solution $\boldsymbol{x}^{\text{LP}}$ by its weighted support graph $G_{\boldsymbol{x}^{\text{LP}}} = (V, E_{\boldsymbol{x}^{\text{LP}}})$ where $E_{\boldsymbol{x}^{\text{LP}}} = \{e \in E : x_e^{\text{LP}} > 0\}$ and a capacity associated with $e \in E_{\boldsymbol{x}^{\text{LP}}}$ is x_e^{LP}. This representation allows the reduction of the domain knowledge requirement and increases the framework's generality.
- *Features of the current node.* We propose 12 statistic features in Table 5.7 to enrich node information in the state representation. The first five features characterize the context of the node within the branch. The remaining features provide information about how effective cuts were at previous nodes. All features are normalized in $[0, 1]$.

5.5.2.3 Discussion

The time-based reward definition provides generality and scalability for the framework, as it can be directly applied to any cut type without needing hyperparameter tuning or domain

Feature group	Feature	Description	Ref.
Node (5)	node_depth	max(1, the node depth / n)	[84]
	cut_node	the number of nodes generating cuts / the number of processed nodes	
	obj_improv	the objective improvement between the current and previous nodes	
	sol_dist	the Euclidean distance between the optimal solutions of the previous and current nodes	
	cut_node_dist	the distance in nodes between the current node and the last generating cut node	
Cuts (7)	nb_curr_cuts	the number of cuts added at the current node	
	curr_cut_dist	the average cut distance at the current node	
	curr_cut_improv	the average objective value improvement obtained by generating cuts at the current node	
	last_cut_dist	the average cut distance at the last generating cut node	
	last_cut_improv	the average objective improvement at the last generating cut node	
	last_round_cut_dist	the average cut distance of the last cut round at the last generating cut node	
	last_round_cut_improv	the objective improvement obtained by the last cut round at the last generating cut node	

Table 5.7. The features extracted from the currently considered node.

knowledge. Additionally, training on the simplified MDP can substantially reduce training time, saving on computational and implementation costs.

However, this reward function relies on the hypothesis that more efficient exploration of the root branch leads to faster solving of the instance overall. This hypothesis is not theoretically guaranteed and may be incorrect in some cases.

5.5.3 Policy parametrization and training

In this subsection, we present the cut evaluator architecture and training, given a reward definition and a state representation.

5.5.3.1 Policy parametrization.

We model the cut evaluator as the Q-value function of the MDP and parameterize it as a neural network with two parts: one embedding states into feature vectors and one approximating action Q-values.

For the first part, we encode state components separately. Graph-structured components (e.g., the LP optimal solution or the problem instance) use GNNs, and statistic feature components use MLPs. The final embedding concatenates component model outputs. Note that this embedding is independent of instance size. After obtaining the state vector, we pass it to the second part, a 3-layer perceptron, to get the Q-value approximation of actions. Figure 5.5 illustrates the cut evaluator architecture.

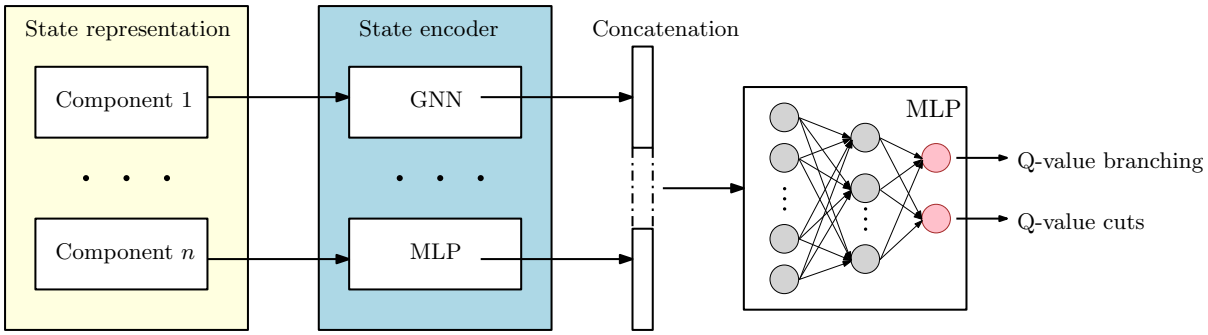


Figure 5.5. The general architecture of the cut evaluator

5.5.3.2 Training algorithm.

To train the cut evaluator, we use the DQN algorithm (Algorithm 5), i.e., the parameters of the cut evaluator are updated to minimize an L2 loss defined with a target network using data sampled from a replay buffer filled with transitions generated during online interactions with the environment. For simplicity, an ϵ-greedy policy is used for exploration.

5.6 Experiments

In this section, we demonstrate the effectiveness of the proposed framework with two well-known combinatorial cut classes: SECs for the TSP and cycle inequalities for the Max-Cut problem.

5.6.1 Setup

All experiments are conducted on a computing server with AMD EPYC 7742 64-core CPU, 504GB of RAM, and an RTX A4000 GPU card with 48GB graphic memory.

5.6.1.1 Problem formulations.

We use IP formulations (5.1) and (5.2) in Section 5.1 to respectively solve the TSP and the Max-Cut problem.

5.6.1.2 Branch-and-cut solver.

We use the commercial solver CPLEX 12.10 as a backend solver and CPLEX UserCutCallback to integrate separation routines for generating cuts into the solver. We keep the CPLEX's default settings, which are expertly tuned. However, to focus on evaluating the benefit of cut generation, we switch off the CPLEX's cuts. The solver time limit is 3600 seconds per instance.

5.6.1.3 Benchmarks.

For the TSP, we use random TSP instances generated following Johnson and McGeoch's generator used for the DIMACS TSP Challenge [82]. These instances are complete graphs. We train on 200 instances with graphs of 200 vertices and evaluate on three instance groups: instances

with graphs of 200 (small), 300 (medium), and 500 vertices (large) and 100 instances per group. Furthermore, we also assess the proposed method on 29 instances with graphs of 200 to 1000 vertices from TSPLIB [46], a well-known library of sample instances for the TSP.

For the Max-Cut problem, we use random instances generated by the well-known rudy generator [83]. These instances are graphs of 10% density. We train on 200 instances with graphs of 100 vertices and also evaluate on three instance groups: instances with graphs of 100 (small), 120 (medium), and 140 (large) vertices. Each group contains 100 instances.

Note that for both problems, the training phase only uses small instances, while the testing phase assesses larger instances. This aims to evaluate the scalability and generalization ability of the proposed framework.

5.6.1.4 The cut detector implementation

To implement the cut detector, we need to determine the node and edge features associated with the separation graph of an LP optimal solution and the form of MP layers.

For SECs, based on their separation routine, we use node degrees as node features and edge capacities as edge features. We use 1-GNN layers proposed in [85] as MP layers. The hidden size of these layers is set to 128.

For cycle inequalities, we empirically observe that the corresponding separation routine can obtain valid cuts in most executions, as shown in Table 1. Thus, we do not use the cut detector for this cut type.

5.6.1.5 The cut evaluator implementation

For SEC generation for the TSP, we set the hyperparameters as follows: the penalty for each action is -0.01; the penalty for each redundant separation routine is -0.1; the bonus for each found cut is 0.01. For cycle inequality generation for the Max-Cut problem, we experimented with many hyperparameter sets, but none of them resulted in a meaningful policy. Thus, we did not apply the gap-based reward function for cycle inequalities. Recall that the state representation corresponding to the gap-based reward function has three components: the LP optimal solution, the problem instance, and the tree features. We encode these with separate models. The LP optimal solution is encoded by a 1-GNN model [85] operating on the separation-purpose graph constructed from the solution. Node degrees are used as node features, and edge capacities as edge features. For the problem instance, we use an MLP layer to embed the 4-dimensional edge features into the same space as the node embeddings. Then, we use a modified GIN architecture [86] to integrate the edge features into updating the node embeddings. For TSP instances, which are complete graphs, we update a node's embedding using its ten nearest neighbors based on edge costs. The tree features are encoded by an MLP with two hidden layers. All feature dimensions are 64.

The time-based reward function's state representation has two components: the LP optimal solution and tree node features. Since we represent the LP solution by a more general graph instead of the separation-purpose graph, we encode this with a Graphormer model [87] combining GNN and Transformer architectures. The tree node features are encoded by an MLP with two hidden layers. Again, all feature dimensions are 64.

5.6.1.6 Training.

We train the cut detector and cut evaluator separately. To train the cut detector for SECs, we generate 96000 labeled fractional solutions from 200 random instances. We train the cut detector within 100 epochs, and the learning rate is 0.0001. For the cut evaluator, we train the Q-learning network with one million steps using the package *stable-baselines3* [88].

5.6.1.7 Baselines.

As in Section 5.3.2, we compare our proposed framework to the skip-factor-based fixed and automatic strategies introduced in [28]. Specifically, these are `FS-1` (generating cuts at every node), `FS-8` (generating cuts at every 8 nodes), and `AS` (generating cuts at every k nodes where k is computed by formulation (5.3)). We experimentally observe that combinatorial cuts are very efficient at the root node but inefficient in the later stages of computation. Therefore, we consistently generate cuts at the root node and stop generating cuts when the IP relative gap is less than 1%, regardless of the strategy used.

5.6.2 The contribution of the cut detector

We first conduct experiments to measure the contribution of the cut detector to the performance. Toward this end, we implement a cut generation strategy `CD` that only uses the cut detector. At every node of the enumeration tree, `CD` generates cuts if the cut detector predicts that some cuts are violated by the corresponding LP optimal solution. Besides, we consider strategies that are combinations of the cut detector and the baselines.

As mentioned earlier, we only use the cut detector for SECs of the TSP, as their separation routine rarely produces valid cuts. Table 5.8 presents the results of the strategies for generating SECs in branch-and-cut for the TSP. `CD+FS-8` and `CD+AS` represent policies where the cut detector is integrated into `FS-8` and `AS`, respectively. The column labeled "*Sepa time (%)*" indicates the proportion of total CPU time spent solving separation problems. Column "*# Preds*" provides information on how many decisions each strategy made. Column "*# Sepa*" gives the number of separation executions, and column "*Sepa cuts (%)*" shows the proportion of these executions obtaining cuts.

As shown in Table 5.8, using the cut detector can improve branch-and-cut performance significantly. The cut detector decreases the separation time, increases the proportion of successful separation executions, and reduces the number of cuts used. On average, the strategies using the cut detector execute the separation routine in approximately 30% of the total decisions made, in which 80% of executions obtain valid cuts.

When comparing the original and cut-detector-used versions of heuristics strategies (i.e., `CD` versus `FS-1`, `CD+FS-8` versus `FS-8`, and `CD+AS` versus `AS`), the cut-detector-used versions usually obtain better results in all aspects including the runtime, separation time and accuracy, and number of cuts used.

Table 5.8 also shows that the impact of cut generation not only depends on the number of separation executions. Indeed, although wasting more time on redundant separation executions than other cut-detector-used strategies, strategy `CD` accelerates performance the most for medium and large instances.

	Strategy	CPU Time	Sepa time (%)	# Preds	# Sepa	Sepa cuts (%)	# Cuts
SMALL	FS-1	109.94	65.4%	119.13	119.13	26.3%	507.0
	FS-8	56.80	38.8%	37.63	37.63	52.7%	493.8
	AS	48.05	21.7%	15.07	15.07	69.9%	439.3
	CD	42.78	38.9%	120.61	27.16	69.4%	**404.2**
	CD+FS-8	**37.70**	24.3%	27.69	11.71	84.8%	432.5
	CD+AS	37.73	**15.25%**	16.84	**7.42**	**93.5%**	425.92
MEDIUM	FS-1	511.10	62.8%	249.21	249.21	25.6%	956.8
	FS-8	424.56	34.5%	77.76	77.76	52.3%	970.2
	AS	440.99	20.3%	30.90	30.90	66.2%	861.7
	CD	**315.16**	37.2%	290.38	60.57	66.1%	**784.6**
	CD+FS-8	318.67	20.9%	61.93	23.76	85.3%	794.5
	CD+AS	316.11	**13.79%**	28.98	**13.86**	**91.5%**	811.55
LARGE	FS-1	2,997.95	56.8%	417.72	417.72	21.7%	2,330.7
	FS-8	2,916.38	26.4%	162.60	162.60	47.4%	2,425.0
	AS	2,978.34	13.0%	77.88	77.88	64.9%	2,265.5
	CD	**2,716.23**	23.3%	700.71	124.71	63.9%	**1,897.3**
	CD+FS-8	2,875.42	11.2%	145.35	54.48	84.2%	2,148.1
	CD+AS	2,935.36	**6.80%**	69.32	**36.21**	**91.1%**	2,171.65
TSPLIB	FS-1	2,115.55	69.2%	855.07	855.07	17.3%	2,446.9
	FS-8	2,056.70	44.5%	373.83	373.83	41.9%	2,694.7
	AS	2,087.74	30.7%	341.83	341.83	59.1%	3,035.1
	CD	**1,915.09**	32.4%	3,755.97	207.55	63.6%	**2,115.6**
	CD+FS-8	1,939.04	24.9%	753.76	103.31	79.9%	2,822.9
	CD+AS	1,940.56	**14.6%**	2,452.24	**64.03**	**89.5%**	2,643.1

Table 5.8. The results of using the cut detector in generating SECs in branch-and-cut for the TSP

5.6.3 The effectiveness of the proposed framework

We now assess the effectiveness of the proposed framework on two combinatorial cut classes: SECs for the TSP and cycle inequalities for the Max-Cut problem. As demonstrated in the experiments conducted in Section 5.1, these cut classes exhibit distinct properties. SECs are infrequent but impactful, whereas cycle inequalities are typically numerous, yet their impact is comparatively weaker. Hence, the designs of the proposed framework for these two cut classes also differ.

5.6.3.1 SEC generation

For SECs, we use both the cut detector and the cut evaluator. We train the cut evaluator with the gap-based (`CE-gap`) and time-based (`CE-time`) reward definitions. Combining with the cut detector, we have two machine-learning-based strategies: `CD+CE-gap` and `CD+CE-time`.

Table 5.9 presents the results of our framework and the baselines on the TSP instances. For each instance group, we report the number of instances that can be solved to optimality within the CPU time limit over the total instances (column *"Solved"*), the average CPU time in seconds (including also the running times of instances that cannot be solved to optimality within the CPU time limit) (column *"CPU Time"*), the average number of nodes in the enumeration tree (column *"# Nodes"*), and the average number of generated SECs (column *"# Cuts"*). Recall that our goal in this paper is to accelerate the branch-and-cut algorithm; thus, the main criterion for comparison is the CPU running time.

As shown in Table 5.9, the strategy `CD+CE-gap` outperforms all the baselines on all instance groups. Indeed, `CD+CE-gap` solves more instances to optimality within a smaller average CPU time. Compared to `FS-8`, `CD+CE-gap` is faster by 5% on average over all random instances, i.e., is 39%, 32%, and 1% faster for small, medium, and large instances, respectively. For the TSPLIB instances, `CD+CE-gap` is faster by 9%, 8%, and 8% compared to `AS`, `FS-8`, and `FS-1`.

In contrast, the `CD+CE-time` strategy exhibits a lower level of competitiveness. One possible explanation could be that this strategy generates too few cuts. In fact, for large and TSPLIB instances, `CD+CE-time` only produces approximately 30% of the average additional cuts compared to other strategies. This low production of cuts is insufficient to strengthen LP relaxations, resulting in poor performance overall.

As predicted, `FS-1` has the smallest tree size on average over all instances, but its running time is the highest due to the extra time spent on generating SECs. On the other hand, too few cuts might be detrimental to the branch-and-cut performance. It can be seen in the comparison between `FS-8` and `AS` strategies on large and medium instances. Indeed, `AS` requires more computing time than `FS-8` despite generating fewer SECs. The numerical results give evidence that our method can balance the separation cost and the benefit of generated SECs.

5.6.3.2 Cycle inequality generation

For cycle inequalities, we do not use the cut detector since most separation executions obtain valid cuts. Furthermore, due to the frequent existence of cycle cuts, we can not find reasonable hyperparameters (i.e., the bonus for found cuts) to train the cut evaluator with the gap-based

	Strategy	Solved	CPU Time	# Nodes	# Cuts
SMALL	FS-1	100/100	109.9	**2,769.0**	506.9
	FS-8	100/100	56.8	3,090.1	493.8
	AS	100/100	48.1	3,521.1	439.3
	CD+CE-gap	100/100	**34.4**	3,185.7	423.7
	CD+CE-time	100/100	46.7	4,769.5	**157.5**
MEDIUM	FS-1	96/100	511.1	**11,969.1**	956.8
	FS-8	98/100	424.6	15,983.4	970.2
	AS	96/100	441.0	26,759.5	861.7
	CD+CE-gap	**99/100**	**288.5**	17,390.6	726.1
	CD+CE-time	96/100	383.84	52,423.7	**545.1**
LARGE	FS-1	32/100	2,998.0	**37,698.9**	2,330.7
	FS-8	35/100	2,916.4	55,882.8	2,425.0
	AS	33/100	2,922.4	71,455.1	2,235.9
	CD+CE-gap	**37/100**	**2,889.7**	72,160.1	1,965.9
	CD+CE-time	29/100	3,015.2	78,291.1	**833.8**
TSPLIB	FS-1	**15/29**	2,062.3	**15,114.4**	2,412.9
	FS-8	14/29	2,056.7	19,797.6	2,694.7
	AS	13/29	2,087.7	23,202.5	2,967.0
	CD+CE-gap	**15/29**	**1,890.1**	30,995.7	2,622.4
	CD+CE-time	14/100	2,212.8	22,995.9	**951.3**

Table 5.9. The numerical results of the SEC generation strategies

reward function. Thus, we only train the cut evaluator with the time-based reward function for cycle inequalities.

Table 5.10 illustrates the efficiency of our approach for cycle inequalities. In the case of small instances, `CE-time` not only minimizes the CPU time but also significantly decreases the number of cuts, roughly by a factor of 2. As for medium instances, this strategy can solve a higher number of instances to optimality within a CPU time that is less than 23% compared to the best among the baseline strategies.

However, the effectiveness of `CE-time` diminishes when applied to large instances. Though it can still solve more instances to optimality in a shorter runtime, the improvement is rather modest. Only a few instances are solved to optimality. A reason could be that the impact of cycle inequalities reaches the limit when applied to large instances, thus requiring more powerful cut types.

5.7 Conclusion

In this chapter, we proposed a data-driven framework to learn cut generation strategies of branch-and-cut algorithms. Our framework comprises two components: the *cut detector* and the *cut evaluator*. We introduced two reward definitions to train the cut evaluator; thus, one can

	Strategy	Solved	CPU Time	# Nodes	# Cuts
SMALL	FS-1	100/100	275.4	**297.4**	14403.4
	FS-8	100/100	105.9	447.8	9107.7
	AS	100/100	87.1	385.9	9959.9
	CE-time	100/100	**60.7**	449.1	**4443.8**
MEDIUM	FS-1	47/100	2777.8	**1666.9**	38111.5
	FS-8	83/100	1703.2	3999.6	**26055.0**
	AS	77/100	1761.2	3671.0	32186.9
	CE-time	**87/100**	**1310.2**	3557.8	30723.8
LARGE	FS-1	0/100	3600.1	**3806.2**	56115.7
	FS-8	1/100	3576.5	5322.2	**45548.5**
	AS	0/100	3600.1	3855.6	56153.8
	CE-time	**2/100**	**3559.2**	4202.8	59108.2

Table 5.10. The numerical results of the cycle inequality generation strategies

easily customize the proposed framework to adapt to the properties of the considered cut type. Experimental results demonstrated that our method outperforms commonly used heuristics (without machine learning) for cut generation. Most importantly, the trained strategies can be applied to instances larger than those used for training.

Our future work is to generalize this framework to other combinatorial cuts of combinatorial optimization problems. Although we have multiple choices for framework architecture, we wonder whether we can only use the cut evaluator with a general-purpose reward function to reduce the expert knowledge and increase the scalability. Another direction is to investigate the use of the proposed framework when branch-and-cut uses more than one cut class.

Conclusion

Branch-and-cut is one of the most powerful methods for solving combinatorial optimization problems exactly. While branch-and-cut has demonstrated its ability to solve computationally challenging problems, its performance is notoriously unstable and strongly dependent on problem properties and implementation details. Within this context, this thesis studies the computational aspects of branch-and-cut in two critical directions of combinatorial optimization: *the fairness of solutions* and *the interaction with machine learning*.

Part I

In the first part of this thesis, we considered branch-and-cut for fair combinatorial optimization, where the fairness of solutions is prioritized. In general, one popular approach to address the issue of fairness is to encode a fairness or unfairness measure in the objective function. These objective functions are usually nonlinear and thus require linearization methods. However, incorporating linearization methods into the problem formulation can cause difficulties in solving it. Part I of this thesis presented ideas, techniques, analyses, and experiments to improve branch-and-cut performance when dealing with nonlinear fair objective functions.

In Chapter 3, we investigated balanced combinatorial optimization, which finds a fair solution by minimizing the difference between the largest and smallest solution components. We provided an empirical example demonstrating the difficulties of using branch-and-cut to solve this problem class. To overcome these challenges, we proposed a specific-purpose branch-and-cut framework whose central component is a new class of local cuts. These cuts utilize information about the enumeration tree to tighten LP relaxations. Combined with additional bounding mechanisms and applied to an NP-hard case study (the BTSP), this framework significantly outperforms general-purpose branch-and-cut algorithms. It also certifies the optimality of more solutions compared to heuristic algorithms for the same test set.

However, the proposed MILP formulation relies on a large constant, known as "big-M," to estimate the smallest solution component. Using big-M can slow branch-and-cut performance and lead to significant losses in numerical precision. One direction for our future work is to develop an alternative MILP formulation of balanced combinatorial optimization that does not require a big-M parameter. We also intend to develop other cut classes to strengthen the linear programming relaxations solved during branch-and-cut to boost algorithmic efficiency. Another direction is to use machine learning to predict the maximum and minimum costs of

the tour and then utilize this information to guide branch-and-cut.

Chapter 4 investigates the second research question, which examines the relationship between two linearization methods for OWA combinatorial problems. Specifically, we have proved the equivalence of the two formulations in terms of their LP relaxations despite their utilization of differing numbers of extra variables. We also conducted experiments applying the two formulations to the OWA TSP to compare their performance within a branch-and-cut algorithm. Interestingly, numerical results showed that the formulation with more variables can be solved more efficiently.

In this chapter, the number of solution components aggregated by the OWA is restricted to be known and fixed; for example, in the OWA TSP, the number of components equals the number of vertices in the graph instance. However, how can we apply this representation of fairness in problems where the number of solution components is unknown, such as the shortest path or minimum cut problem? Additionally, how can we linearize and solve such problems efficiently using branch-and-cut? Besides, we also want to construct approximation algorithms to solve OWA combinatorial optimization.

Part II

In Part II of this thesis, we studied the intersection of machine learning and combinatorial optimization. Machine learning offers immense potential to revolutionize the way combinatorial optimization problems are solved today. It can automatically learn problem-specific heuristics to substitute manually designed and generic rules for crucial decision problems within traditional methods. The second part of this thesis initiates an exploratory study to exploit machine learning's capacity to enhance branch-and-cut algorithms, answering our research question 3.

Chapter 5 presents the first method of using machine learning for cut generation, a vital decision problem in branch-and-cut, with a focus on combinatorial cuts. Our method employs a combination of supervised and reinforcement learning to train models to decide whether to generate cuts or to branch at a node in the branch-and-bound tree. Whereas traditional methods heavily rely on experts' theoretical and empirical knowledge about the properties of combinatorial cuts, the proposed method in Chapter 5 can automatically extract this information from data. Furthermore, once trained, the model can perform on arbitrary-sized instances. Experiments on two well-known classes of combinatorial cuts have shown impressive results: Our method can solve more instances to optimality faster than hand-crafted heuristics. Although the method's generality is limited for large-sized instances, this work has demonstrated machine learning's potential to address branch-and-cut inner problems like cut generation.

While machine learning has shown potential in this area, there is still significant room for further improvement. Firstly, while we have two reward definitions to adapt to various properties of combinatorial cuts, these definitions still rely on the choice of hyperparameters and hypotheses, which reduce the generality and scalability of the framework. Thus, it is worth exploring an alternative definition of the reward function that is more general. Secondly, since representing the entire enumeration tree is too complex, we used a substitute representation that only utilizes a subset of tree information. This can limit the full potential of machine learning for studying generation strategies. Designing better representations of the search tree is

a promising approach to improving the framework. Finally, investigating the theoretical impact of cuts generated at a node on the overall performance of branch-and-cut is an interesting direction we want to pursue.

In a bigger plan, we want to build a more versatile framework that can decide between generating cuts or branching, select the branching variables if branching, choose which cuts to add if generating cuts, and determine which node to explore next.

Résumé en français

L'optimisation combinatoire est un sous-domaine de l'optimisation mathématique qui consiste à trouver une solution optimale à partir d'un ensemble fini de possibilités. Il s'agit de l'un des domaines de recherche les plus actifs de ces dernières années puisque des milliers de problèmes de décision réels qui ont un impact significatif sur notre vie quotidienne peuvent être formulés sous forme de problèmes d'optimisation combinatoire. Les exemples incluent l'optimisation des plans de ressources dans le domaine de la santé [3], la minimisation des coûts de transport globaux d'un calendrier de livraison [4], la minimisation des émissions de gaz à effet de serre des opérations logistiques [5] et la maximisation du profit total du projet sélection de portefeuille dans l'entreprise [6]. Ces problèmes sont souvent caractérisés par leur complexité et leur difficulté, nombre d'entre eux étant classés comme NP-difficiles, ce qui représente des défis restant à relever pour parvenir à des solutions optimales, voire quasi optimales.

Bien que les solutions possibles d'optimisation combinatoire soient limitées, leur nombre augmente de façon exponentielle avec la taille de l'instance, rendant la recherche exhaustive insoluble en un temps reasonnable. Pour surmonter ce défi, les chercheurs ont développé des algorithmes de recherche efficaces adaptés à des problèmes d'optimisation combinatoire spécifiques. Cependant, ces algorithmes nécessitent généralement une compréhension approfondie des caractéristiques du problème et un investissement important en temps et en ressources. Une approche alternative consiste à établir des méthodes génériques pour résoudre divers problèmes d'optimisation combinatoire.

Une de ces méthodes génériques est la Programmation Linéaire en Nombre Entier Mixte (MILP), une technique puissante pour modéliser des problèmes. La plupart des problèmes d'optimisation combinatoire peuvent être formulés naturellement sous forme de formulations MILP où les décisions sont représentées par des variables pouvant prendre des valeurs continues ou discrètes. Les relations entre ces variables sont définies par des contraintes linéaires, et le but est d'optimiser une fonction objectif linéaire soumise à ces contraintes.

Une fois qu'un problème d'optimisation combinatoire est formulé comme un problème MILP, il peut être résolu à l'aide de solveurs MILP. La méthode de base des solveurs MILP modernes de pointe est *le branch-and-bound et le plan de coupe*, qui fusionne deux méthodes bien connues : le branch-and-bound et le plan de coupe. Alors que le branch-and-bound et le plan de coupe divisent récursivement l'espace de recherche en sous-espaces plus petits et débornent la fonction objectif dans chaque sous-espace, la méthode du plan de coupe ajoute de manière itérative des inégalités valides pour affiner l'espace de recherche. La combinaison de

ces deux méthodes permet au branch-and-cut d'élaguer efficacement l'espace de solution et de converger rapidement vers la solution optimale.

Les performances empiriques branch-and-cut sont irrégulières et sensibles aux attributs du problème et des détails de mise en œuvre. Par exemple, un changement mineur dans la fonction objectif ou dans la manière de représenter les contraintes peut entraîner une grande différence des performances. Des facteurs de mise en œuvre, comme les stratégies de sélection internes, l'utilisation d'heuristiques ou les choix d'algorithmes de programmation linéaire (LP), peuvent également avoir un impact sur l'efficacité du branch-and-cut.

Motivée par ces problématiques, cette thèse vise à combler le manque de connaissances entourant les aspects informatiques du branch-and-cut lors de la résolution de problèmes d'optimisation combinatoire.

Dans cette thèse, nous nous concentrons sur l'étude du branch-and-cut dans le contexte de deux dimensions critiques de l'optimisation combinatoire : *l'équité des solutions* et *l'intégration de l'apprentissage automatique*.

Equité des solutions

L'équité est un concept fondamental qui intrigue les humains depuis des siècles. C'est quelque chose que nous désirons et pour lequel nous aspirons tous, car c'est le fondement de nos interactions et relations sociales. L'idée d'équité ne se borne pas à un contexte ou une situation spécifique ; il s'agit plutôt d'un concept universel qui apparaît dans tous les domaines de la vie. Par exemple, en temps de crise, il est essentiel d'assurer la répartition équitable de ressources rares, telles que la nourriture et les médicaments, afin que chaque individu ait accès aux nécessités de la vie. Dans un autre contexte, comme dans une salle de classe, tous les élèves devraient être traités de manière uniforme et sans discrimination fondée sur des facteurs tels que leur richesse, leurs capacités ou leur apparence.

De par sa nature, la question de l'équité a également reçu une attention considérable de la part des chercheurs en optimisation combinatoire. Ce problème se pose naturellement dans les problèmes d'optimisation combinatoire. Par exemple, dans le problème d'affectation qui consiste à attribuer des tâches aux travailleurs, une solution dans laquelle certains travailleurs reçoivent des charges de travail nettement plus importantes que d'autres pourrait être perçue comme un traitement injuste.

Diverses approches ont été proposées dans la littérature pour aborder la question de l'équité dans l'optimisation combinatoire avec différentes manières de modéliser l'équité. Dans cette thèse, nous nous concentrons sur deux approches populaires : *optimisation combinatoire équilibrée* [7] et *optimisation combinatoire moyenne pondérée ordonnée (OWA)* [8].

Optimisation combinatoire équilibrée

Optimisation combinatoire équilibrée, proposée par Martello et al. [7], trouve une solution équitable en minimisant la différence de valeurs entre les composants les plus chers et les moins chers ; nous appelons cette différence *la distance max-min*, pour abréger. Cette approche a d'abord été définie dans le contexte du problème d'affectation [7] puis a été étendue

et généralisée à d'autres cas particuliers d'optimisation combinatoire en raison de sa nature intuitive [7, 9–19].

Il existe deux approches principales pour résoudre l'optimisation combinatoire équilibrée. La première consiste à construire directement une solution avec la plus petite distance max-min en exploitant des structures spécifiques au problème. Cette approche s'est principalement concentrée sur des problèmes solvable par des polynômes. Le second est basé sur l'algorithme à double un cadre itératif général trouvant les seuils des composants les plus grands et les plus petits. Ces seuils peuvent être trouvés en vérifiant à plusieurs reprises l'existence d'une solution réalisable dans un ensemble donné via ce que l'on appelle *sous-programme de faisabilité*. Par conséquent, la complexité de cet algorithme dépend du sous-programme de faisabilité, qui est parfois NP-difficile. Les variantes visent à réduire la complexité des problèmes de vérification de faisabilité et le nombre d'itérations nécessaires.

À notre connaissance, aucun algorithme générique basé sur MILP n'a été proposé pour une optimisation combinatoire équilibrée, bien que MILP ait réussi à résoudre un large éventail de problèmes. La raison en est que le principal défi lors de la résolution d'une optimisation combinatoire équilibrée ne consiste pas à formuler des problèmes mais à déborner les composants les plus grands et les plus petits. Lorsque ces composants ne sont pas étroitement liés, il devient beaucoup plus difficile d'élaguer les nœuds de l'arbre d'énumération pendant le processus de branch-and-cut. Par conséquent, résoudre des problèmes d'optimisation combinatoire équilibrée par branch-and-cut peut être incroyablement long et inefficace.

Question de recherche 1 : *Comment concevoir le branch-and-cut pour résoudre efficacement l'optimisation combinatoire équilibrée ?*

Pour répondre à cette question, au chapitre 3, nous proposons une algorithme de branchement et de coupe pour une optimisation combinatoire équilibrée. Le point central de notre algorithme est une nouvelle classe de plans de coupe locaux appelés *coupes locales de délimitation*. Ces plans de coupe ne nécessitent aucune structure spécifique au problème et exploitent plutôt les informations sur les arbres de branchement et de coupe pour mieux borner le plus petit coût de composant. Ainsi, ils peuvent être appliqués à tous les problèmes d'optimisation combinatoire équilibrée.

Nous évaluons l'algorithme proposé sur un cas NP-difficile d'optimisation combinatoire équilibrée, c'est-à-dire le problème du voyageur de commerce équilibré (BTSP), qui n'a été résolu que de manière heuristique par [13] dans la littérature. Pour améliorer encore les performances, nous développons un algorithme de borne inférieure pour initialiser une borne inférieure sur la valeur optimale BTSP. Nous proposons également un algorithme de recherche locale pour fournir une borne supérieure initiale et améliorer la solution existante lors du branchement et de la coupe. De plus, nous introduisons des techniques d'élimination des variables et de fixation de variables pour réduire la taille du problème et resserrer les relaxations de programmation linéaire (LP). Des expériences sur le même jeu d'instances TSPLIB montrent que notre approche peut résoudre 63 instances sur 65 avec une optimalité prouvée, tandis que l'heuristique basée sur le double seuil [13] ne certifie l'optimalité que de 27 instances.

Optimisation combinatoire Ordered Weighted Average

Notez que l'optimisation combinatoire équilibrée se concentre uniquement sur la distance max-min, ce qui peut conduire à des solutions inefficaces concernant le résultat total. Une approche plus sophistiquée est *l'optimisation combinatoire OWA* [8], dont la fonction objectif intègre l'opérateur OWA [20]. L'intuition derrière l'optimisation combinatoire OWA découle d'une observation selon laquelle, lorsqu'il s'agit d'équité de solution, nous nous soucions de l'ensemble des valeurs des composants sans considérer quel valeur spécifique prend chaque composante. Ainsi, cette approche considère chaque composante comme un objectif individuel et regroupe les objectifs de l'opérateur OWA. Les solutions d'optimisation combinatoire OWA sont efficaces au sens Pareto-optimal, prenant en compte la minimisation des inégalités selon l'approche Pigou-Dalton. Cette méthode basée sur OWA est bien adaptée aux situations réelles nécessitant des compromis appropriés entre équité et efficacité.

A cause de la fonction objectif, les problèmes d'optimisation combinatoire OWA sont non linéaires, même si leurs contraintes d'origine sont linéaires. Heureusement, il existe deux méthodes de linéarisation de l'OWA avec des poids décroissants. La première a été proposée par Orgyczak et al. [1], qui reformule l'OWA comme une combinaison de composants de Lorenz et représente chaque composant de Lorenz comme un dual d'un programme linéaire. La seconde a été introduite par Chassein et al. [2], qui utilise, comme valeur OWA, le permutaedre permettant d'obtenir le maximum parmi toutes les permutations du produit interne entre les poids OWA et les composants de la solution. En termes de taille, la méthode de Chassein utilise moins de variables que celle d'Orgyczak. De plus, lorsqu'elle est appliquée à plusieurs problèmes d'optimisation continue, la formulation de Chassein peut être résolue plus rapidement [2]. Cependant, dans l'optimisation combinatoire OWA, l'intégration de méthodes de linéarisation dans la formulation du problème peut entraîner des difficultés supplémentaires. Par conséquent, des questions demeurent sur la comparaison et la relation entre les deux méthodes d'optimisation combinatoire OWA.

Question de recherche 2 : *Y a-t-il une relation entre les deux méthodes de linéarisation pour l'optimisation combinatoire OWA ? La formulation avec moins de variables est-elle toujours résolue plus rapidement par branch-and-cut ?*

Dans le chapitre 4, notre objectif est de comparer les deux formulations MILP lors de leur incorporation dans le branchement et la coupe. Nous proposons des comparaisons sous les aspects théoriques et empiriques, en particulier

- Nous prouvons que les deux formulations sont équivalentes en termes de relaxation de programmation linéaire.
- Nous estimons la qualité de la valeur OWA de la solution optimale de l'optimisation combinatoire classique.
- Nous évaluons les performances du branch-and-cut lors de l'utilisation de ces formulations pour résoudre le problème du voyageur de commerce (TSP) d'OWA. Bien que O-MILP utilise plus de variables supplémentaires que C-MILP, il peut être résolu plus rapidement que C-MILP sur OWA TSP.

Apprentissage automatique pour l'optimisation combinatoire

Branch-and-cut est hautement configurable et ses performances dépendent fortement de la configuration des stratégies de décision internes, telles que la sélection de variables, la sélection de nœuds, la sélection de coupes ou la génération de coupes. Une configuration habile peut aider à résoudre des problèmes de calcul difficiles. Cependant, identifier une configuration efficace est un défi difficile.

Dans des situations pratiques, nous résolvons généralement de manière répétée un problème d'optimisation combinatoire avec de nombreuses instances différentes mais liées. Ce processus génère une vaste quantité de données historiques pouvant contenir des motifs significatifs. Une idée naturelle consiste à exploiter ces motifs pour résoudre plus rapidement de nouvelles instances. Cependant, l'extraction manuelle de ces motifs nécessite un effort expert important. Par conséquent, un outil automatisé est nécessaire pour découvrir et exploiter systématiquement des motifs permettant d'élaborer des politiques décisionnelles habiles.

Récemment, l'apprentissage automatique est apparu comme un outil prometteur pour apprendre automatiquement des stratégies efficaces de branchements et de coupes à partir d'expériences de résolution passées. Les algorithmes d'apprentissage automatique peuvent identifier des modèles dans les données et améliorer les performances dans divers domaines. Par rapport aux heuristiques de réglage manuel, l'apprentissage automatique présente deux avantages clés. Premièrement, cette approche est plus systématique, ce qui pourrait conduire à des heuristiques plus efficaces. Deuxièmement, l'apprentissage automatique permet de développer rapidement des stratégies spécialisées, même pour des classes de problèmes nouvelles et peu familières, en s'appuyant moins sur les connaissances du domaine humain.

Tirant parti de ces avantages, les chercheurs ont appliqué l'apprentissage automatique pour apprendre des politiques pour les problèmes de décision de branchements et de coupes, y compris la sélection de variables [21,22], la sélection de nœuds [23] et la sélection de coupes [24], et obtenu des résultats prometteurs. Cependant, il reste un problème important qui n'a pas encore été entièrement résolu : la *génération de coupe*, un défi de conception clé lors de la combinaison des méthodes de branch-and-cut et de plan de coupe. La génération de coupes fait référence au problème de décider s'il faut générer des coupes ou créer une branche aux nœuds de l'arbre de branchement et de liaison pour réduire le temps d'exécution global. Ce problème a un impact considérable sur les performances de branchement et de coupe. En effet, générer des coupes peut réduire considérablement la taille des arbres mais peut également endommager le temps d'exécution global en raison de la résolution des problèmes de séparation et des relaxations LP.

De plus, les travaux antérieurs d'apprentissage automatique sur les problèmes de décision liés aux coupes se sont concentrés principalement sur les coupes à usage général, basées sur les conditions d'intégralité des variables. Les coupes combinatoires, une classe cruciale de coupes codant pour des structures spécifiques à un problème, n'ont pas encore reçu beaucoup d'attention. Le manque d'études sur la génération de coupes combinatoires constitue une lacune notable dans la recherche, car les coupes combinatoires sont indispensables pour résoudre l'optimisation combinatoire, en particulier les problèmes NP-difficiles.

Question de recherche 3 : *Comment pouvons-nous utiliser l'apprentissage automatique pour apprendre une politique de génération de coupes combinatoires ?*

Dans le chapitre 5, nous proposons un cadre général basé sur l'apprentissage automatique pour apprendre des stratégies de génération de coupes combinatoires. Notre idée est de formuler la génération de coupes sous la forme d'un problème de décision markovien (MDP) et de former un agent à apprendre les politiques.

Bien que l'idée de représenter les problèmes de décision de branchement et de coupe sous forme de MDP ne soit pas nouvelle [22,23], notre approche aborde le problème d'apprentissage d'une manière nouvelle à travers deux contributions :

- Contrairement à la plupart des travaux antérieurs qui utilisent l'apprentissage par imitation pour imiter un expert précieux mais coûteux, nous formons l'agent par apprentissage par renforcement avec des fonctions de récompense soigneusement conçues. Pour s'adapter à la variété des propriétés des coupes combinatoires, nous introduisons deux définitions de fonctions de récompense : une basée sur la différence relatif de MILP et une autre basée sur le temps d'exécution. Nous proposons en outre plusieurs techniques de mise en forme des récompenses pour accélérer la convergence de l'algorithme.

- Notre méthode utilise une approche hybride intégrant à la fois un apprentissage supervisé et par renforcement. Ce choix de conception est dû à l'observation selon laquelle des prédictions coûteuses ne devraient être tentées qu'aux nœuds prometteurs où la résolution des problèmes de séparation peut obtenir des coupes. En particulier, notre framework se compose de deux composants distincts : 1) un *cut detector*, un réseau neuronal graphique qui détecte l'existence de coupes, et 2) un *cut évaluateur*, un agent d'apprentissage par renforcement qui sélectionne entre génération des coupes et des ramifications. Diviser la génération de coupes en tâches spécialisées nous permet d'utiliser des modèles légers tout en garantissant la précision.

Grâce aux choix de conception, notre cadre est polyvalent et applicable à divers problèmes d'optimisation combinatoire avec différents types de coupes. De plus, les politiques formées sont adaptables à des instances de taille arbitraire, même si elles sont initialement formées avec des exemples de taille fixe. Pour évaluer l'efficacité des méthodes proposées, nous menons des expériences sur deux problèmes d'optimisation combinatoire NP-difficiles bien connus : le TSP avec contraintes d'élimination de sous-tours et le problème de coupe maximale avec inégalités de sous-tours. Les résultats expérimentaux démontrent que nos méthodes peuvent améliorer considérablement les performances des algorithmes de branchement et de coupe.

List of Figures

List of Tables

Acronyms

BTSP balanced traveling salesman problem. 3

DT double-threshold. 46

GGI Generalized Gini Index. 51

GNN graph neural network. 18

IB iterative bottleneck. 46

IP integer programming. 7

LP linear programming. 2, 95

MDP Markov decision process. 19

MILP mixed-integer linear programming. 1, 94

MLP multilayer perceptron. 15

MP message passing. 18

OWA ordered weighted average. 2, 95

RL reinforcement learning. 19

SGD stochastic gradient descent. 17

Bibliography

[1] Włodzimierz Ogryczak and Tomasz Śliwiński. On solving linear programs with the ordered weighted averaging objective. *European Journal of Operational Research*, 148(1):80–91, 2003. (Cited on pages ix, 3, 48, 49, 53, and 97.)

[2] André Chassein and Marc Goerigk. Alternative formulations for the ordered weighted averaging objective. *Information Processing Letters*, 115(6-8):604–608, 2015. (Cited on pages ix, 4, 48, 49, 53, 54, 56, 62, and 97.)

[3] Mohammad Fattahi, Esmaeil Keyvanshokooh, Devika Kannan, and Kannan Govindan. Resource planning strategies for healthcare systems during a pandemic. *European Journal of Operational Research*, 304(1):192–206, 2023. (Cited on pages 1 and 94.)

[4] Paolo Toth and Daniele Vigo. *Vehicle routing: problems, methods, and applications*. SIAM, 2014. (Cited on pages 1 and 94.)

[5] Rommert Dekker, Jacqueline Bloemhof, and Ioannis Mallidis. Operations research for green logistics–an overview of aspects, issues, contributions and challenges. *European journal of operational research*, 219(3):671–679, 2012. (Cited on pages 1 and 94.)

[6] Vahid Mohagheghi, Seyed Meysam Mousavi, Jurgita Antuchevičienė, and Mohammad Mojtahed. Project portfolio selection problems: a review of models, uncertainty approaches, solution techniques, and case studies. *Technological and Economic Development of Economy*, 25(6):1380–1412, 2019. (Cited on pages 1 and 94.)

[7] Silvano Martello, William R Pulleyblank, Paolo Toth, and Dominique De Werra. Balanced optimization problems. *Operations Research Letters*, 3(5):275–278, 1984. (Cited on pages 2, 26, 27, 28, 95, and 96.)

[8] Viet Hung Nguyen and Paul Weng. An efficient primal-dual algorithm for fair combinatorial optimization problems. In *International Conference on Combinatorial Optimization and Applications*, pages 324–339. Springer, 2017. (Cited on pages 2, 3, 50, 60, 61, 62, 95, and 97.)

[9] Paolo M. Camerini, Francesco Maffioli, Silvano Martello, and Paolo Toth. Most and least uniform spanning trees. *Discrete Applied Mathematics*, 15, 1986. (Cited on pages 2, 27, and 96.)

[10] Zvi Galil and Baruch Schieber. On finding most uniform spanning trees. *Discrete Applied Mathematics*, 20, 1988. (Cited on pages 2, 26, 27, and 96.)

[11] Naoki Katoh and Kazuo Iwano. Efficient algorithms for minimum range cut problems. *Networks*, 24, 1994. (Cited on pages 2, 26, 27, and 96.)

[12] Ravindra K. Ahuja. *European Journal of Operational Research*. (Cited on pages 2, 27, and 96.)

[13] John Larusic and Abraham P Punnen. The balanced traveling salesmanproblem. *Computers & Operations Research*, 38(5):868–875, 2011. (Cited on pages 2, 3, 28, 33, 42, 43, 45, 46, 47, and 96.)

[14] Annette Ficker, Frits Spieksma, and Gerhard J. Woeginger. Balanced optimization with vector costs. *SSRN Electronic Journal*, 2016. (Cited on pages 2 and 26.)

[15] Paolo M Camerini, Francesco Maffioli, Silvano Martello, and Paolo Toth. Most and least uniform spanning trees. *Discrete Applied Mathematics*, 15(2-3):181–197, 1986. (Cited on pages 2, 26, and 96.)

[16] Anna GRINČOVÁ, Daniela KRAVECOVÁ, and Marcel KUDLÁČ. Alternative approach to data network optimization. *Acta Electrotechnica et Informatica No*, 6(1):2, 2006. (Cited on pages 2, 26, and 96.)

[17] CW Duin and A Volgenant. Minimum deviation and balanced optimization: A unified approach. *Operations Research Letters*, 10(1):43–48, 1991. (Cited on pages 2, 26, and 96.)

[18] Abraham P Punnen and Yash P Aneja. Lexicographic balanced optimization problems. *Operations Research Letters*, 32(1):27–30, 2004. (Cited on pages 2, 26, and 96.)

[19] Lara Turner, Abraham P Punnen, Yash P Aneja, and Horst W Hamacher. On generalized balanced optimization problems. *Mathematical Methods of Operations Research*, 73(1):19–27, 2011. (Cited on pages 2, 26, and 96.)

[20] Ronald R Yager. On ordered weighted averaging aggregation operators in multicriteria decisionmaking. *IEEE Transactions on systems, Man, and Cybernetics*, 18(1):183–190, 1988. (Cited on pages 3, 51, and 97.)

[21] Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *AAAI*, 2016. (Cited on pages 4, 65, 70, 71, and 98.)

[22] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 2019. (Cited on pages 4, 65, 70, 71, 98, and 99.)

[23] He He, Hal Daume III, and Jason M Eisner. Learning to search in branch and bound algorithms. *Advances in neural information processing systems*, 2014. (Cited on pages 4, 65, 70, 71, 98, and 99.)

[24] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut. In *ICML*. PMLR, 2020. (Cited on pages 4, 65, 70, 71, and 98.)

[25] George Dantzig. *Linear programming and extensions*. Princeton university press, 1963. (Cited on page 8.)

[26] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311, 1984. (Cited on page 8.)

[27] Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275 – 278, 1958. (Cited on page 9.)

[28] Egon Balas, Sebastian Ceria, Gérard Cornuéjols, and N Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1–9, 1996. (Cited on pages 11 and 86.)

[29] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning internal representations by error propagation, 1985. (Cited on page 16.)

[30] Jack Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, pages 462–466, 1952. (Cited on page 17.)

[31] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991. (Cited on page 17.)

[32] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6, 2005. (Cited on page 22.)

[33] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015. (Cited on page 22.)

[34] Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. *Machine learning*, 49:209–232, 2002. (Cited on page 23.)

[35] Lara Turner. Variants of shortest path problems. *Algorithmic Oper. Res.*, 6, 2011. (Cited on page 26.)

[36] Paola Cappanera and Maria Grazia Scutella. Balanced paths in acyclic networks: Tractable cases and related approaches. *Networks: An International Journal*, 45(2):104–111, 2005. (Cited on page 26.)

[37] Maria Grazia Scutellà. A strongly polynomial algorithm for the uniform balanced network flow problem. *Discrete applied mathematics*, 81(1-3):123–131, 1998. (Cited on page 26.)

[38] Štefan Berežnỳ and Vladimír Lacko. Balanced problems on graphs with categorization of edges. *Discussiones Mathematicae Graph Theory*, 23(1):5–21, 2003. (Cited on page 26.)

[39] AP Punnen and KPK Nair. Constrained balanced optimization problems. *Computers & Mathematics with Applications*, 37(9):157–163, 1999. (Cited on page 26.)

[40] Lacko Vladimír Berežný, Štefan. The color-balanced spanning tree problem. *Kybernetika*, 41(4):[539]–546, 2005. (Cited on page 26.)

[41] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 114–122, 1981. (Cited on page 27.)

[42] John E Hopcroft and Richard M Karp. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973. (Cited on page 28.)

[43] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955. (Cited on page 28.)

[44] John LaRusic and Abraham P Punnen. The asymmetric bottleneck traveling salesman problem: algorithms, complexity and empirical analysis. *Computers & Operations Research*, 43:20–35, 2014. (Cited on page 28.)

[45] Esther M Arkin, Yi-Jen Chiang, Joseph SB Mitchell, Steven S Skiena, and Tae-Cheon Yang. On the maximum scatter traveling salesperson problem. *SIAM Journal on Computing*, 29(2):515–544, 1999. (Cited on page 28.)

[46] Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991. (Cited on pages 28, 30, 42, 61, 67, and 85.)

[47] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954. (Cited on pages 29, 53, and 67.)

[48] Shen Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269, 1965. (Cited on pages 33 and 37.)

[49] Keld Helsgaun. General k-opt submoves for the lin–kernighan tsp heuristic. *Mathematical Programming Computation*, 1(2):119–163, 2009. (Cited on pages 33 and 37.)

[50] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972. (Cited on page 36.)

[51] Ralph E Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961. (Cited on pages 42 and 67.)

[52] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008. (Cited on page 42.)

[53] Włodzimierz Ogryczak, Tomasz Śliwiński, and Adam Wierzbicki. Fair resource allocation schemes and network dimensioning problems. *Journal of Telecommunications and Information Technology*, (3):34–42, 2003. (Cited on page 49.)

[54] Lucie Galand and Olivier Spanjaard. Exact algorithms for owa-optimization in multiobjective spanning tree problems. *Computers & Operations Research*, 39(7):1540–1554, 2012. (Cited on page 49.)

[55] Wlodzimierz Ogryczak, Patrice Perny, and Paul Weng. On minimizing ordered weighted regrets in multiobjective markov decision processes. In *Algorithmic Decision Theory: Second International Conference, ADT 2011, Piscataway, NJ, USA, October 26-28, 2011. Proceedings 2*, pages 190–204. Springer, 2011. (Cited on page 49.)

[56] John A Weymark. Generalized gini inequality indices. *Mathematical Social Sciences*, 1(4):409–430, 1981. (Cited on pages 49 and 51.)

[57] Julien Lesca, Michel Minoux, and Patrice Perny. The fair owa one-to-one assignment problem: Np-hardness and polynomial time special cases. *algorithmica*, 81:98–123, 2019. (Cited on page 49.)

[58] Arthur Cecil Pigou. *Wealth and welfare*. Macmillan and Company, limited, 1912. (Cited on page 51.)

[59] Hugh Dalton. The measurement of the inequality of incomes. *The Economic Journal*, 30(119):348–361, 1920. (Cited on page 51.)

[60] G.H. Hardy, J.E. Littlewood, and G. Pólya. *Inequalities*. Cambridge Mathematical Library. Cambridge University Press, 1952. (Cited on page 51.)

[61] Max O Lorenz. Methods of measuring the concentration of wealth. *Publications of the American statistical association*, 9(70):209–219, 1905. (Cited on page 51.)

[62] David B Shmoys and David P Williamson. Analyzing the held-karp tsp bound: A monotonicity property with application. *Information Processing Letters*, 35(6):281–285, 1990. (Cited on page 60.)

[63] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017. (Cited on page 65.)

[64] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1):185–195, 2017. (Cited on pages 65, 70, and 71.)

[65] Elias B Khalil, Bistra Dilkina, George L Nemhauser, Shabbir Ahmed, and Yufen Shao. Learning to run heuristics in tree search. In *Ijcai*, pages 659–666, 2017. (Cited on pages 65 and 70.)

[66] Antonia Chmiela, Elias Khalil, Ambros Gleixner, Andrea Lodi, and Sebastian Pokutta. Learning to schedule heuristics in branch and bound. *NeurIPS*, 34:24235–24246, 2021. (Cited on pages 65 and 70.)

[67] Zeren Huang, Kerong Wang, Furui Liu, Hui-Ling Zhen, Weinan Zhang, Mingxuan Yuan, Jianye Hao, Yong Yu, and Jun Wang. Learning to select cuts for efficient mixed-integer programming. *Pattern Recognition*, 123:108353, 2022. (Cited on pages 65, 70, and 72.)

[68] Max B Paulus, Giulia Zarpellon, Andreas Krause, Laurent Charlin, and Chris Maddison. Learning to cut by looking ahead: Cutting plane selection via imitation learning. In *ICML*, 2022. (Cited on pages 65, 70, 72, and 74.)

[69] Manfred W Padberg and Saman Hong. *On the symmetric travelling salesman problem: a computational study*. Springer, 1980. (Cited on page 67.)

[70] Francisco Barahona and Ali Ridha Mahjoub. On the cut polytope. *Mathematical programming*, 36:157–173, 1986. (Cited on pages 68 and 69.)

[71] Rinaldi Giovanni. Rudy. `https://biqmac.aau.at/biqmaclib.html`, 1998. (Cited on page 69.)

[72] Manfred Padberg and Giovanni Rinaldi. Branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 1991. (Cited on page 70.)

[73] Egon Balas, Sebastián Ceria, and Gérard Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework, 1996. (Cited on pages 70 and 74.)

[74] Tobias Achterberg. *Constraint integer programming*. PhD thesis, 2007. (Cited on page 70.)

[75] Franz Wesselmann and U Stuhl. Implementing cutting plane management and selection techniques. In *Technical Report*. University of Paderborn, 2012. (Cited on page 70.)

[76] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2021. (Cited on page 70.)

[77] Marc Etheve, Zacharie Alès, Côme Bissuel, Olivier Juan, and Safia Kedad-Sidhoum. Reinforcement learning for variable selection in a branch and bound algorithm. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 176–185. Springer, 2020. (Cited on page 71.)

[78] Jialin Song, Ravi Lanka, Albert Zhao, Aadyot Bhatnagar, Yisong Yue, and Masahiro Ono. Learning to search via retrospective imitation. *arXiv preprint arXiv:1804.00846*, 2018. (Cited on page 71.)

[79] Abdel Ghani Labassi, Didier Chételat, and Andrea Lodi. Learning to compare nodes in branch and bound with graph neural networks. *Advances in Neural Information Processing Systems*, 35:32000–32010, 2022. (Cited on page 71.)

[80] Zhihai Wang, Xijun Li, Jie Wang, Yufei Kuang, Mingxuan Yuan, Jia Zeng, Yongdong Zhang, and Feng Wu. Learning cut selection for mixed-integer linear programming via hierarchical sequence model. *arXiv preprint arXiv:2302.00244*, 2023. (Cited on page 72.)

[81] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. *Finding cuts in the TSP (A preliminary report)*, volume 95. Citeseer, 1995. (Cited on page 74.)

[82] DS Johnson and LA McGeoch. Benchmark code and instance generation codes. http://dimacs.rutgers.edu/archive/Challenges/TSP/download.html, 2002. (Cited on pages 74 and 84.)

[83] Rudy generator. https://web.stanford.edu/~yyye/yyye/Gset/rudy.c. (Cited on pages 74 and 85.)

[84] Giulia Zarpellon, Jason Jo, Andrea Lodi, and Yoshua Bengio. Parameterizing branch-and-bound search trees to learn branching policies. In *AAAI*, volume 35, pages 3931–3939, 2021. (Cited on pages 80 and 83.)

[85] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *AAAI*, volume 33, pages 4602–4609, 2019. (Cited on page 85.)

[86] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. *arXiv preprint arXiv:1905.12265*, 2019. (Cited on page 85.)

[87] C Ying, T Cai, S Luo, S Zheng, G Ke, D He, Y Shen, and TY Liu. Do transformers really perform bad for graph representation? arxiv 2021. *arXiv preprint arXiv:2106.05234*. (Cited on page 85.)

[88] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. (Cited on page 86.)