Algorithms for Scheduling and Routing Problems

by

Kamyar Khodamoradi

M.Sc., Sharif University of Technology, 2008
B.Sc., Azad University, Tehran North Branch, 2006

Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Science

SIMON FRASER UNIVERSITY
Fall 2016

Approval

Name:	**Kamyar Khodamoradi**
Degree:	**Doctor of Philosophy (Computing Science)**
Title:	***Algorithms for Scheduling and Routing Problems***
Examining Committee:	**Chair:** Faraz Hach Research Associate

Ramesh Krishnamurti
Senior Supervisor
Professor _____

Binay Bhattacharya
Supervisor
Professor _____

Pavol Hell
Supervisor
Professor _____

Leonid Chindelevitch
Internal Examiner
Assistant Professor _____

Yash Aneja
External Examiner
Professor
Odette School of Business
University of Windsor _____

Date Defended: 29 November 2016

Abstract

Optimization has been a central topic in most scientific disciplines for centuries. Continuous optimization has long benefited from well-established techniques of calculus. Discrete optimization, on the other hand, has risen to prominence quite recently. Advances in combinatorial optimization and integer programming in the past few decades, together with the improvement of computer hardware have enabled computer scientists to approach the problems in this area both theoretically and computationally. However, obtaining the exact solution for many discrete optimization problems remains is still a challenging task, mainly because most of these problems are **NP**-hard. Under the widespread assumption that $\mathbf{P} \neq \mathbf{NP}$, these problems are intractable from a computational complexity standpoint. Therefore, we should settle for near-optimal solutions. In this thesis, we develop techniques to obtain solutions that are provably close to the optimal for different indivisible resource allocation problems. Indivisible resource allocation encompasses a large class of problems in discrete optimization which can appear in disguise in various theoretical or applied settings.

Specifically, we consider two indivisible resource allocation problems. The first one is a variant of the vehicle routing problem known as *Skill Vehicle Routing* problem, in which the aim is to obtain optimal tours for a fleet of vehicles that provides service to a set of customers. Each of the vehicles possesses a particular set of skills suitable for a subset of the tasks. Each customer, based on the type of service he requires, can only be served by a subset of vehicles. We study this problem computationally and find either the optimal solution or a relatively tight bound on the optimal solution on fairly large problem instances. The second problem involves approximation algorithms for two versions of the classic scheduling problem, the *restricted* $R||C_{\max}$ and the *restricted Santa Claus* problem. The objective is to design a polynomial time approximation scheme (PTAS) for *ordered* instances of the two problems. Finally, we consider the class of precedence hierarchies in which the neighborhoods of the processors form Laminar families. We show similar results for a generalization of this model.

Keywords: Vehicle routing problem; Integer linear programming; Branch-and-cut algorithms; Prize-collecting traveling salesman problem; Scheduling problems; Convex and bipartite permutation graphs

To Sofia, Mohammadyar, and Kimya...

...and in memory of
Sahabali Fazli

Acknowledgements

First and foremost, I would like to thank my advisor, Ramesh Krishnamurti, for his unwavering support throughout my years at SFU, his patient mentorship, and his invaluable insight in combinatorial optimization. I am for always indebted to Ramesh for everything he has taught me. I would also like to thank Binay Bhattacharya for his encouragements and immense wisdom he shared with me over many pleasant conversations.

My sincere thanks go to:

My mom and dad, for giving me their boundless love and motivation to do my best, and my sister, for her infinite kindness and support.

Pavol Hell, for sharing his vast knowledge of algorithmic graph theory.

Arash Rafiey, for being a friend and a mentor to me in my long journey at SFU.

Abraham Punnen, for giving me the best advice and pointing me in the right direction whenever I had doubts in a project.

My great collaborators, Thomas Sauerwald, Georgios Stamoulis, Alexandre Stauffer, Peter Kling, and Petra Berenbrink. I have learned a lot from working with them.

Yash Aneja and Leonid Chindelevitch, for kindly agreeing to be my examiners. Their insightful comments and feedback were crucial for improving the quality of this thesis.

My brilliant colleagues and comrades in the Theory lab at SFU throughout these years, especially Ehsan Iranmanesh, Shenwei Huang, Ante Custic, Vladyslav Sokol, Ali Ershadi, Carrie Wang, Oren Shklarsky, Sita Gakhar, and Ali Pazooki.

Many great friends I was fortunate enough to have here in Vancouver, including Hamed Yaghoubi, Mohammad Tayebi, Faraz Hach, Amir Hedayaty, Amir Aavani, Shahab Tasharofi, Fereydoun Hormozdiari, Farhad Hormizdiari, Pashootan Vaezi, Navid Imani, Iman Sarrafi, Iman Hajirasouliha, Laleh Samii, Sara Ejtemaee, Zahra Ehtemam, Arghavan Ahmadi, Ali Khalili, Amir Yaghoubi, Akbar Rafiey, Hassan Khosravi, Mehran Khodabandeh, and Amirali Sharifian. The list goes on.

Contents

List of Tables

List of Figures

Part I

The Framework

Chapter 1

Resource Allocation Problems

1.1 Indivisible Resource Allocation

Resource Allocation problem is the problem of distributing a set of scarce resources among some customers (or players) while satisfying some constraints. It has applications in areas such as on-line auctions, scheduling, production planning, spectrum allocation, and load balancing, and therefore is considered to be of tremendous practical importance. At the same time, the problem is also interesting from a theoretical perspective as finding a solution to a Resource Allocation problem is usually challenging and calls for mathematically involved approaches. Due to its twofold nature, the problem has occupied a central role in many fields of study such as computer science, operations research, game theory, economics, social choice theory, and mathematics. The approaches taken for tackling resource allocation problems can widely vary based on whether the resource in question can be split into any arbitrary fraction or not. Consequently, two subcategories of the problem arise: Divisible Resource Allocation and Indivisible Resource Allocation. As the names imply, in the former case any fraction of a resource can be allocated to players, whereas in the latter, a resource is assigned to a player in its entirety. Divisible Resource Allocation problem, most commonly referred to as the Cake Cutting problem, is a problem of interest for many theoretical and empirical disciplines. Many research papers in these disciplines focus on *fair division* of limited divisible resources. Also along this line, much work has been done on defining the meaning and perception of fairness, and conditions that can either guarantee the existence of a fair division or imply its absence. The long list of such disciplines includes economics, game theory, mathematics, sociology, political sciences, philosophy, and psychology [16]. We shall not discuss the Cake Cutting problem any further and refer the interested reader to a seminal book by Brams and Taylor on the topic [16].

On the other hand, the indivisible case has more of a combinatorial flavour to it and thus has mostly been a topic for the algorithmic line of research [5]. For this very reason, it

is the primary focus of this research. In the most basic form, an instance of a Indivisible Resource Allocation problem, or IRA for short, includes the following components:

(1) a set $\mathcal{P}$ of m *players*

(2) set of $\mathcal{R}$ of n indivisible *resources* or equivalently, items

(3) a value function or *utility function* $f_i : 2^{\mathcal{R}} \to \mathbb{R}$ for each player $i \in \mathcal{P}$

(4) an objective function on any arbitrary allocation of items

where an allocation is a partitioning of the items $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m)$ such that $\cup_{i=1}^{m} \mathcal{A}_i = \mathcal{R}$, and $\mathcal{A}_i \cap \mathcal{A}_k = \emptyset$ whenever $i \neq k$. Each of the sets $\mathcal{A}_i$ is to be allocated to a player $i \in \mathcal{P}$. Note that some of the sets $\mathcal{A}_i$ might be empty. It has to mentioned that an instance of an IRA problem may encompass many other components beyond the four mentioned above. For example, every resource might be only available in a certain time window, or some constraints can be inflicted on the problem based on some specific job characteristics. We will discuss these additional components in further detail in Section 1.3.1. The goal then is to find the allocations $\mathcal{A}$ in such a way the objective function is "optimized" (we will formally define the optimality criterion in Section 1.3.1), while the constraints of the problem instance are all satisfied.

The Indivisible Resource Allocation (IRA) research pursues three targets: i) efficient algorithms that optimize the objective function when possible, ii) spell out conditions under which finding an optimal allocation is a hard task (meaning that no polynomial algorithm seems to exist to find such allocations), and iii) find solutions that are as close as possible to optimal allocation.

In this thesis, we study two problems, either of which has its connections to the Indivisible Resource Allocation problem. These problems are variations of the Vehicle Routing problem and the Scheduling problem. In what follows, our objective is to place the Vehicle Routing and Scheduling problem in the context of resource allocation. Then, we survey the general literature on the IRA and introduce some basic notations and definition along the way.

1.2 Connections to Vehicle Routing

We propose that the concept of Indivisible Resource Allocation is connected to the class of vehicle routing problems in at least two ways. First, roughly speaking, vehicle routing problems deal with situations where some customers demand a particular service. The objective of the problem is to assign vehicles to subsets of customers to meet their needs. In most of the settings, the vehicles cannot be fractionally assigned to a job, and every customer must be covered by at least one vehicle. These requirements are the very constraints that shape any Indivisible Resource Allocation problem. On top of that, of course, the solution

must take other restrictions into account. For instance, the customers are located in a metric space, and the cost of an assignment must reflect the distances between these locations. Also, we seek tours that start and end at a particular place, say a depot. These are the type of constraints that feature in the large family of Travelling Salesman Problem. In that light, we can view many variations of the vehicle routing as a hybrid of an IRA problem and a TSP problem, in which the indivisible resources are the vehicles and the costs are dictated by rules similar to those of the TSP.

The second way that connects the vehicle routing to IRA problems is the underlying model on which the solution techniques are based. Through experience, we have come across mathematical formulations that can help in solving both problems efficiently. Such formulations prove to be useful for both problems, whether the intention is to solve the problem for large instances computationally or to provide approximation algorithms with desirable approximation guarantees. We will make references to these formulations in time.

1.3 Connections to Scheduling

For long, the notion of resource allocation has been tied to the classic problem of scheduling. Scheduling, in short, is best described as "*optimal allocation of scarce resources to activities over time*" [79]. The connection can easily be made between the two subjects when one casts a scheduling problem as a particular type of resource allocation, in which the resources can represent CPU processing times one wishes to allocate to some tasks. Because of this close connection, we will borrow from the rich jargon of machine scheduling for our discussion of resource allocation problems whenever the equivalence of the resource allocation instance and the scheduling instance of the problem is apparent. We will discuss this further in the next subsection.

1.3.1 A Standard Notation

As mentioned before, when we are referring to an instance of an IRA problem, we use $\mathcal{P}$ to represent the set of m players and $\mathcal{R}$ to denote the set of n resources. Also, it is assumed in many cases that any resource $j \in \mathcal{R}$ has a value v_{ij} to a player $i \in \mathcal{P}$. If any arbitrary resource has the same value to all players, we let v_j denote that value for ease of notation. In the context of job scheduling, however, which constitutes a significant portion of the indivisible resource allocation literature, the players are usually thought of as machines, and the resources correspond to jobs to be scheduled on these machines. Therefore, in such settings, we switch to a different notation. We let $\mathcal{M}$ denote the set of m machines, $\mathcal{J}$ the set of n jobs, and p_{ij} the processing time of a job $j \in \mathcal{J}$ on a machine $i \in \mathcal{M}$. Equivalently, if jobs have the same processing times on all machines (which occurs in certain types of the Machine Scheduling problem), we shorten the processing times to p_j for all jobs $j \in \mathcal{J}$.

A General Framework

Still, a lot has to be clarified before we can define a solid instance of the MACHINE SCHEDULING problem. But even after doing so, the number of problem types that may arise is virtually unlimited. Therefore, our discussion of the problem types has to be selective. Consequently, we will focus on a particular type of resource allocation which is closely related to single-criterion *deterministic machine scheduling*. The term "single-criterion" reflects the fact that the optimization is carried out under a single optimality criterion, and the term "deterministic" indicates that all the information that define a problem instance are known in advance with certainty. To define an instance of the MACHINE SCHEDULING problem, we will use a general framework which was first introduced by Graham et al. [54]. This general framework spans a large class of scheduling problems, as a result of which we need to narrow down our focus even further to a finer subclass of the large family of scheduling problems. In what follows, we will only mention parts of the framework that apply to the subclass in consideration. At the end of this chapter, it should be clear to the reader what the problems in the aforementioned subclass exactly are.

Graham et al. propose that an instance of MACHINE SCHEDULING problem is defined by the machine environment, the job characteristics, and the optimality criterion. Base on these three features, they introduce a three-field classification scheme in the form of $\alpha \,|\, \beta \,|\, \gamma$. We will discuss each field in what follows.

- **Machine environment (α).** We let $\alpha = \alpha_1\, \alpha_2$. For the sake of our discussion, we allow α_1 to to assume a value from the set $\{\circ, P, Q, R\}$ [1]. The symbol $\circ$ denotes the empty symbol and is reserved for the case of a single machine here. The other three values are characterized as follows:

  - $\alpha_1 = P$: The machines are *identical* parallel machines, meaning that each job has the same running time on all the machines, or $p_{ij} = p_j$ for all jobs $j \in \mathcal{J}$.
  - $\alpha_1 = Q$: The machines are *uniform* parallel machines, meaning that each machine $i \in \mathcal{M}$ has a speed s_i associated with it, and each job $j \in \mathcal{J}$ has a fixed processing time p_j on a machine with unit speed. Therefore, the processing time of job j on machine i will be $p_{ij} = p_j/s_i$.
  - $\alpha_1 = R$: The machines are *unrelated* parallel machines, meaning that a job $j \in \mathcal{J}$ has a machine-dependent processing time p_{ij} on a machine $i \in \mathcal{M}$.

  For the case where there exists only one machine ($\alpha_1 = \circ$), we set $\alpha_2 = 1$. In all other cases, α_2 can either be $\circ$, meaning that m, the number of jobs, is a part of the input (and thus a variable), or it takes a positive integer value, which means that m is a constant equal to α_2.

[1] Other notations are available for α_1 which can represent a variety of scheduling problems such as *open shops*, *flow shops*, and *job shops*.

- **Job characteristics (β).** Job characteristics may include some restrictions on how jobs are scheduled on the machines. For instance, whether or not job preemption is allowed, or whether certain precedence relations rule the order in which different jobs can be fed into the machines. Throughout our entire discussion, we let $\beta = \circ$, representing the most general case where no such restrictions exist.

- **Optimality criterion (γ).** For the third field, the most common criteria studied in the field are of one of these two forms: $\{f_{max}, \sum f_i\}$, where f is the value function. If the optimality criterion is set to f_{max}, it indicates that we seek to minimize the maximum of f_i among all machines $i \in \mathcal{M}$, and if it is set to $\sum f_i$, it means we intend to minimize the sum of all f_i's. Note that we are using the MACHINE SCHEDULING-specific notation we introduced earlier. For many variants of the IRA problem, which in a way are the conceptual duals of scheduling problems, the goal would be to maximize f_{min} or $\sum f_i$. A couple of options for the choice of f are listed below (Note that f_i's are associated with machines).

  - C: Completion time. The time by which a job is processed on machine i.
  - L: Lateness. In case jobs come with e predefined deadline, the difference between the completion time of a job on machine i and its deadline.

Example 1.3.1. *If the goal is to minimize the maximum completion time of a set of unrelated machines, we let $\alpha = R$, $\beta = \circ$, and $\gamma = C_{max}$, therefore, we refer to such a problem as an $R||C_{max}$ problem.*

1.3.2 Social Welfare and Fairness

The problems we consider in this thesis have two types of objectives: i) to optimize *social welfare*, or ii) to ensure some level of *fairness*. We will briefly discuss each of these objectives in what follows.

Optimizing the social welfare is probably one of the most natural objectives one can pursue when dealing with a resource allocation problem. The challenge is to allocate indivisible resources to players in such a way that the sum of the utilities of all players is maximized. In the context of scheduling, resource values are replaced by job processing times. Therefore it makes more sense to set the objective to minimize the sum of all utilities, the utility of a machine being its workload in this case. One can approach the problem of maximizing the social welfare from a game theoretic point of view, in which the research splits into two major sub-categories. One takes a more existential route and involves identification of the equilibria of the game, and comparing them against outcomes (allocations) that would be socially optimal. Most of the results in this area are not constructive, and we will not survey any of them. The other sub-category deals with a more constructive approach by aiming at the design of algorithms that can achieve a socially optimum allocation. More

specifically, one can look at a setting in which players have (private or public) valuations for subsets of resources instead of valuing items directly. This particular type of valuation functions gives rise to the combinatorial nature of the problem. The goal is to develop a procedure for eliciting these valuations from the players and use them to decide how to allocate the resources to players in a way that the social benefit is optimized. This viewpoint of resource allocation is very suitable for modeling auctions, in which players bid on bundles of items using predefined communication rules (sometimes referred to as the *bidding language*), and after a period has passed, the auctioneer selects the winner of the auction. Consequently, this class of problems is referred to as *combinatorial auctions* in algorithmic game theory literature. We can consider the problem under the assumption of perfect or imperfect information. In the former, our job is to design exact or approximation algorithms that gather information from the bidders through some well-defined queries and decide on how to allocate the items so as to maximize the social welfare. The former falls into the domain of truthful mechanism design. In this area, we devise simple procedures for the auction that ensure an optimal (or approximately optimal if absolute optimality is not feasible) social welfare provided that all players bid truthfully. In other words, the social welfare is maximized only when the players reveal their real private valuations. Some of the most remarkable results on social welfare maximizing truthful mechanisms "involve multiple assumptions on how players would react to various incentives" [39], as they need more sophisticated models to tame the complexity of the problem. Due to these complications and space considerations, we avoid all the results on this topic.

Welfare-maximizing resource allocation algorithms are not suitable in settings where some notion of fairness needs to be guaranteed as there might be instances in which all the resources are allocated to a single player in the optimal solution [94]. Therefore, an effort has been made to formulate and solve the problem under some constraints to ensure fairness and the result is a subclass of the original problem known as the Fair IRA problem. Researchers have proposed several criteria for fairness. Among the most prominent ones is the concept of *Max-Min* (or *Min-Max*) allocations adopted from scheduling literature, and *envy-free* allocations which has it roots in non-cooperative games. In the former, we seek to maximize the utility of the player who has received the minimum value set of resources (or in the dual context of scheduling minimize the maximum workload among all machines), thus enforcing fairness to the system by ensuring a minimum level of benefit for all players. In the latter, similar to the treatment of social welfare maximizing problems, two independent subcategories are considered: one under the assumption of perfect information, and the other assuming otherwise. Regardless, we wish to minimize the tendency of players to prefer the set of resources allocated to other players over their bundle in both these subcategories. Envy-free allocations will be defined precisely in Section 1.4.3.

1.4 Resource Allocation Variations

In this section, we first mention a few definitions regarding the utility functions that we frequently use in later sections. What comes after serves the primary intention of the chapter, which is to briefly survey some of the most significant variations of the Fair IRA and Welfare Maximizing IRA problems.

1.4.1 Preliminaries

As one can expect, the utility functions of the players is a major determinant of the problem type, which directly affects the complexity of the resource allocation setting. Since the early days of the resource allocation research, theoreticians of this field have tried to propose functions that can closely mimic the utility of players of a real world setting. As a result, many different types of utility functions have been the subject of this study throughout the years. We list some of the most prominent in this section.

Definition 1 (Monotone Utility Functions)**.** *A utility function f is said to be monotone if and only if $f_i(C) \leqslant f_i(D)$ for every subset of items $C \subseteq D \subseteq \mathcal{R}$ and every player $i \in \mathcal{P}$.*

Monotonicity is modeled after a quite natural assumption that if you add more resources to a bundle already assigned to a player without removing any items, they will not end up unhappier than before. Another natural utility function is defined as follows:

Definition 2 (Additive Utility Function)**.** *A utility function f is said to be additive if and only if for every player $i \in \mathcal{P}$ and every subset of items $C \subseteq \mathcal{R}$, $f_i(C) = \sum_{j \in C} f_i(\{j\})$.*

One might argue that additivity is more of a simplification than a natural property of the utility function. A fine reasoning for this argument is that of the *marginal values*. For any player who has already received a hefty bundle of resources, the added happiness caused by a single new resource to her bundle is usually insignificant, while the very same item could make another player much happier if they had received only a small or empty set. This following definition best captures the mentioned phenomena.

Definition 3 (Submodular Utility Functions)**.** *The following definitions are equivalent:*

- *A utility function f is said to be submodular if and only if for subsets of items $C, D \subseteq \mathcal{R}$ with $C \subseteq D$ and $j \in \mathcal{R} \setminus D$, $f_i(C \cup \{j\}) - f_i(C) \geqslant f_i(D \cup \{j\}) - f_i(D)$ for every player $i \in \mathcal{P}$.*
- *A utility function f is said to be submodular if and only if for subsets of items $C, D \subseteq \mathcal{R}$, $f_i(C) + f_i(D) \geqslant f_i(C \cup D) + f_i(C \cap D)$ for every player $i \in \mathcal{P}$.*

Among other types of the utility function that we are going to deal with in the section are the class two of *fractionally sub-additive* utility functions and *sub-additive* utility functions defined below.

Definition 4 (Fractionally Sub-additive Utility Functions)**.** *A utility function f is said to be fractionally sub-additive if and only if for every subsets of items $C, D_1, D_2, \ldots, D_k$, $f(C) \leqslant \sum_k \alpha_k f(D_k)$ with $0 \leqslant \alpha_k \leqslant 1$ for all k whenever the sets D_k form a "fractional cover" [39] for C, or more precisely, whenever the following holds: for every resource $j \in C$, $\sum_{k \mid D_K \ni j} \alpha_k \geqslant 1$.*

Definition 5 (Sub-additive or Complement-free Utility Functions)**.** *A utility function f is said to be sub-additive or* complement-free *if and only if for every subsets of items C and D, $f(C \cup D) \leqslant f(C) + f(D)$.*

1.4.2 Welfare Maximizing Indivisible Resource Allocation

Combinatorial Auctions

Auctions come in a variety of forms. There are numerous ways of determining the valuation functions as well as choosing the bidding language[2], Furthermore, different auctions have different governing rules such as the number of rounds, the winner determination criteria, how to charge the winner or winners of the auction, and so forth. As a result, the notion of combinatorial auctions has a vast reach that encompasses a myriad of problems and spans over many disciplines other than theoretical computer science such as operations research and economics. Although the topic is relatively recent and continues to grow rapidly, the literature on combinatorial auctions is already so rich that any comprehensive survey can easily amount up to a book[3]. Naturally, we will only look at a modest subset of combinatorial auctions which are closely related to scheduling and resource allocation, namely those that are carried out in just one round with the objective of maximizing the social welfare.

A Word on the Bidding Language

The bidding language is a protocol of communication that specifies how the players or bidders in the auction may present their bids to the auctioneer. The importance of such a specification becomes evident in combinatorial auctions due to the underlying complexity of the bids. For a set $\mathcal{R}$ of resources of cardinality n, there are $2^n - 1$ non-empty subsets, but for obvious reasons it would not be practical to specify a bid in this setting with $2^n - 1$ real numbers, one valuation for each non-empty subset. The goal of a bidding language is to encode the bids more succinctly. Ignoring the unnecessary technicalities, we let the bidding language be a mapping from the set of possible bids to a set of finite strings of characters. In what follows, we summarize some of the main aspects of the bidding languages that we will use in later sections. For more information on bidding languages, we refer the interested reader to chapter 9 of [27].

[2]Roughly speaking, the bidding language is the formal representation of the bids in a combinatorial auction. We will have a short discussion on bidding languages in the next subsection.

[3]In fact, there are books already published on the topic by notable authors (see [27] as an example).

The "basic" bidding languages are as follows:

- **Atomic Bids**. The most basic form of bids are the atomic bids. Each bidder can submit a pair (C, v) in which C is a subset of items (or resources), and v is the valuation, or the amount that the bidder is willing to pay for that subset. It means that only the subset C is of interest to the bidder, and all other subsets of items are of value 0 to her. These types of bids have a very limited expressive power of course as they cannot even capture the additive valuation of two items.

- **OR Bids**. In these types of bids, each bidder can submit an arbitrary number of subset-value pairs, $(C_1, v_1), (C_2, v_2), \ldots, (C_k, v_k)$, in which each of the tuples is an atomic bid, and C_i's are disjoint for $i = 1, 2, \ldots, k$. A bid of this kind can be represented by $(C_1, v_1)\ OR\ (C_2, v_2)\ OR\ \ldots\ OR\ (C_k, v_k)$. Here, the bidder is willing to acquire any number of the subsets for which she has submitted a pair for a price equal to the sum of their respective v_i's. It is known that OR bids can represent valuations over items that don't have any *substitutability*, i.e., those where for all $C \cap D = \varnothing$, $f(C \cup D) \geqslant f(C) + f(D)$, and only them, in which f is the utility (or valuation) function.

- **XOR Bids**. Here, each bidder can submit an arbitrary number of subset-value pairs (atomic bids), $(C_1, v_1), (C_2, v_2), \ldots, (C_k, v_k)$, and is willing to acquire only one of the the sets C_i for the price of v_i. It can be proved that the XOR bids can represent any possible valuation over the sets of resources. An XOR bid is represented by $(C_1, v_1)\ XOR\ (C_2, v_2)\ XOR\ \ldots\ XOR\ (C_k, v_k)$.

Remark 1.4.1. *Let the size of a bid be the number of its atomic bids. Assume a setting in which the bidders have additive utilities over a set of n resources. Then, their valuations can be represented by OR-bids of size n, while to show such a valuation using XOR bids one needs bids of size 2^n. Therefore, neither OR-bids alone nor XOR-bids by themselves are suitable to represent most of the valuation functions under study in the literature. One way to overcome this shortcoming is by combining the two types.*

One can also think of a combination of OR and XOR bids in the two following ways:

- **OR-of-XOR Bids**. As the name suggests, each bidder can submit an arbitrary number k of bids, and she is willing to acquire any number of such bids. The only difference is that here, any of these k bids is an XOR-bid instead of an atomic one. This class of bids in sometimes referred to as OXS-bids.

- **XOR-of-OR Bids**. Likewise, each bidder can submit an arbitrary number k of OR-bids in this model, and she is interested in acquiring only one of such bids. Bids of this type are often called XOS-bids in combinatorial auctions literature.

No matter the choice of the bidding language, there are instances of the problem for which there is no escape from an exponential representation of the valuations [90]. To overcome this issue, we assume the existence of some query oracle who can answer the given query in constant time. Of course, this assumption can make a tremendous computational power available for the algorithm designer, and weaken the model extensively. Therefore, to avoid oversimplification of the problem, the type of queries answered by the oracle must be chosen with care. The most popular types of oracles studied in the literature are as follows:

- **Value Oracles**. A value oracle can return the value of any bundle of resources for a player i. Value oracles are more prevalent in computer science literature.
- **Demand Oracles**. Demand oracles are stronger models which are typical of a more economic point of view. A demand oracle can answer vector queries of the type $(v_1, v_2, \ldots, v_n)$ for a given bidder i, one valuation for each item in $\mathcal{R}$. The answer returned for such a query is a subset of resources $\mathcal{T} \subseteq \mathcal{R}$ for which $f_i(\mathcal{T}) - \sum_{i=1}^{n} v_i$ is maximized. In other words, the oracle returns a bundle of items most demanded by a player i for a set of given prices $v_1, \ldots, v_n$.

Remark 1.4.2. *Blumrosen and Nisan show in [14] that demand oracles can simulate value oracles. Therefore, one can readily use any algorithm designed for the value oracle model in a demand oracle model as well, but not the other way around.*

We survey models using both of these types of oracles in the next sections.

Types of Combinatorial Auctions

Different variations of combinatorial auctions can be identified based on their input utility functions (valuation functions used by the bidders). Some of these types include:

(1) Sub-additive or Complement-free auctions

(2) Fractionally Sub-additive auctions

(3) Submodular auctions

(4) XOS auctions

(5) OXS functions

These classes of auctions are not mutually exclusive, and in fact, they form a hierarchy. Lehmann et al. propose such a hierarchy of combinatorial auctions in [81]: OXS $\subset$ Submodular $\subset$ XOS $\subset$ Complement-free. Furthermore, Feige shows in [39] that the class of fractionally sub-additive utility functions (and the corresponding type of auctions) is the same as the class of XOS utility functions (auctions). In this section, we mainly focus on the top three types of auctions in the hierarchy as they are the most appealing to the

combinatorial auctions researchers as well. Naturally, any of the results for a higher class in the hierarchy also holds for the lower level ones, but not the other way around.

Combinatorial Auctions with Submodular Bidders

- **Utility functions:** submodular
- **Objective function:** find an allocation $\mathcal{A}$ that maximizes the sum of utilities over all players

For the case of submodular bidders, Lehmann et al. provide a 2-approximation greedy algorithm under the value oracle model [81]. A series of papers improve this factor to $\frac{e-1}{e}$. First, Dobzinski and Schapira provided a $2 - \frac{1}{n}$-approximation in [34] using a randomized algorithm. Later, Fleischer et al. [45] extended the $\frac{e-1}{e}$ approximation to a special case of submodular bidders in which each of the bidders has a budget constraint as well by rounding a linear programming relaxation. Then, Călinescu et al. provide a $\frac{e-1}{e}$ approximation algorithm in [18] for a subclass of submodular utility functions, namely when f, the utility function, is the sum of weighted rank functions of matroids. They apply the *pipage rounding* technique of Ageev and Sviridenko [1] to a linear programming formulation of the problem to obtain their main result. Finally, Vondrak achieves a randomized (expected) $(\frac{e-1}{e} - o(1))$-approximation algorithm for submodular bidder in the value oracle model in [106]. This result heavily depends on the pipage rounding technique and its adaptation to matroid polytope by Călinescu et al. in [18]. As for the lower bounds on approximability of the problem, Khot et al. prove that it is **NP**-hard to approximate the social welfare within a factor of $\frac{e-1}{e} + \varepsilon$ [73].

For the stronger model of demand oracles, Dobzinski and Schapira provide a $\frac{e-1}{e}$-approximation algorithm [34]. Their technique involves solving a linear programming relaxation of the problem and using randomized rounding to find a "pre-allocation", and later finalize the allocation in such a way that the approximation factor is guaranteed. This bound is later improved to $\frac{e-1}{e} + \varepsilon$ (in expectation) for small values of ε ($\varepsilon \approx 0.01$) by Feige and Vondrak in [40]. Also, Chakrabarty and Goel show an inapproximability result of $\frac{15}{16} + \varepsilon$ in [22], unless $\mathbf{P} = \mathbf{NP}$.

Combinatorial Auctions with XOS Bidders

- **Utility functions:** fractionally sub-additive
- **Objective function:** find an allocation $\mathcal{A}$ that maximizes the sum of utilities over all players

As mentioned before, this class is the same as auctions with bidders who have fractional sub-additive utility functions. For this case, and under the value oracle model, Dobzinski,

Nisan, and Schapira give a $(1/\sqrt{m})$-approximation algorithm [33][4]. The algorithm is indeed provided for the broader class of complement-free bidders, which works for the case of XOS bidders as well. Also, Dobzinski and Schapira show in [34] that it is NP-hard to approximate the problem within an approximation factor better than $\frac{1}{m^{1/4}}$.

The problem has also been studied under the demand oracle model. A deterministic $\frac{1}{2}$-approximation algorithm is given in [33]. Dobzinski and Schapira also provide a randomized $\frac{e-1}{e}$-approximation algorithm for the problem that uses a slightly different oracle called the *XOS oracle* [5], which can answer XOS queries [34]. Feige later improves this result using a different rounding technique to get the same $\frac{e-1}{e}$ approximation factor for the more general case of demand queries [39]. Furthermore, Dobzinski et al. show that it is NP-hard to approximate the social welfare within a factor of $\frac{e-1}{e} + \varepsilon$ for the demand oracle model [33].

Combinatorial Auctions with Complement-free Bidders

- **Utility functions:** sub-additive
- **Objective function:** find an allocation $\mathcal{A}$ that maximizes the sum of utilities over all players

The class of complement-free bidders corresponds to those auction in which the bidders have sub-additive valuation functions. Using the value oracle, the authors of [33] provide a $(1/\sqrt{m})$-approximation algorithm for the problem. The factor of $\frac{1}{\sqrt{m}}$ is the best approximation guarantee known for this problem. As for the lower bound on approximability, Dobzinski and Schapira prove NP-hardness of approximating the social welfare in this case within a factor of $\frac{1}{m^{1/4}}$ [34]. This lower bound is improved to $\frac{1}{2} + \varepsilon$ by Feige [39] for any arbitrarily small yet positive real value of ε.

For the model of demand oracles, Dobzinski et al. have a $(1/\log m)$-approximate solution [33]. Similar to the [34], an LP-relaxation of the problem is solved and randomly rounded to find a pre-allocation, which is later improved to obtain the final allocation with the desired approximation guarantee. The linear programming used here is often referred to as the *configuration LP*. We explain this formulation in detail in Section 1.4.3 when we discuss the General Santa Claus problem. Later, Feige provides a randomized $\frac{1}{2}$-approximation for this class [39]. He obtains this resul by again finding a pre-allocation through rounding the solution to an LP relaxation of the problem and then improving the rounding in the next step. This is what Feige calls "two stage rounding" in [39]. In the first round, a tentative allocation is produced which may assign an item to more than one player. This is regarded as a competition among the players who have the same item, each fighting over its possession.

[4] This result also appears in a conference version of the paper published in the proceedings of STOC 2005.

[5] XOS oracles can be considered a special case of demand oracles. They behave in the same was as the demand oracles in general, but instead of taking one valuation for each item, they accept an XOS bid and return the subset of items that maximizes this bid.

The second rounding stage is then the contention removal round, which tries to allocate each shared item to a player who has the maximum marginal utility for it. The author of this thesis is not aware of any inapproximability results for the case of complement-free bidders and demand oracles.

1.4.3 Fair Indivisible Resource Allocation

Envy-free Allocation of Indivisible Resources

Lipton et al. considered the ENVY-FREE IRA in [85]. Generally speaking, an allocation of items to players is *envy-free*, if every player thinks she has the portion of items with the largest value (based on her own utility function), and thus, does not envy any other player. A more formal statement of this concept is given in Definition 6.

Definition 6 (Envy-free Allocation)**.** *[85] An allocation of items $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m)$ to m players is envy-free if every player prefers her own share than the share of any other player. That is, for each $i \in \mathcal{P}$, $argmax_k\{f_i(\mathcal{A}_k)\} = \mathcal{A}_i$.*

Also, the envy for a specific allocation $\mathcal{A}$ is defined as follows.

Definition 7. *[85] The envy of a player i for another player k in an allocation $\mathcal{A}$ is defined as*

$$e_{ik}(\mathcal{A}) = \max\{0, f_i(\mathcal{A}_k) - f_i(\mathcal{A}_i)\}.$$

For the entire allocation $\mathcal{A}$, the envy denoted by $e(\mathcal{A})$ is defined as

$$e(\mathcal{A}) = \max_{i,\, k \in \mathcal{P}} \{e_{ik}(A)\}.$$

For indivisible goods, it is not always possible to find envy-free allocations. Therefore, a natural question to ask is what is the minimum envy attainable in a given setting. Researchers have defined the following problems based on this question.

Bounded Envy Indivisible Resource Allocation

- **Utility functions:** monotone
- **Objective function:** find an allocation $\mathcal{A}$ with a bounded envy $e(\mathcal{A}) \leqslant \alpha$, for some given upper bound α

Lipton et al. prove that for the case of indivisible items, an allocation with envy at most α always exists, where alpha is the maximum *marginal utility* [85]. The marginal utility of a player is the maximum increase in her utility after adding one more item to her bundle.

More formally,

$$\alpha = \max_{C,i,j}\{f_i(C \cup \{j\}) - f_i(C)\}$$

where $C \subset \mathcal{R}$ is a bundle of items, $i \in \mathcal{P}$ is a player, and $j \in \mathcal{R}$ is a single item. They also provide an algorithm that finds such allocations in $\mathcal{O}(nm^3)$, where m is the number of players and n is the number of items.

Minimum Envy and Minimum Envy-ratio Problems

Although the Bounded Envy IRA guarantees the existence of an allocation with envy bounded by the maximum marginal utility, in many cases the minimum attainable envy can be even smaller. Lipton et al. approached the problem from an optimization perspective, aiming at minimizing the envy for indivisible resources [85]. They studied two types of envy minimization algorithms. The utility functions are the same for both settings, and the only difference is in the objective function.

The Minimum Envy IRA problem is defined as follows:

- **Utility functions:** monotone or additive, accessible through an oracle
- **Objective function:** find an allocation $\mathcal{A}$ that minimizes the envy defined as $\max_{i,k}\{0, f_i(A_k) - f_i(A_i)\}$

and the Minimum Envy-ratio IRA is defined in the following manner:

- **Utility functions:** monotone or additive, accessible through an oracle
- **Objective function:** find an allocation $\mathcal{A}$ that minimizes the envy ratio defined as $\max_{i,k}\{1, \frac{f_i(A_k)}{f_i(A_i)}\}$

As mentioned in the utility functions, the existence of a *value oracle* is assumed in this model. Any arbitrary algorithm should make queries to the oracle about the value of a subset of items for an individual player to access the utility functions.

For the case of monotone utility functions, Lipton et al. show that any deterministic algorithm that solves Minimum Envy IRA or Minimum Envy-ratio IRA runs in a number of rounds that is exponential in the number of items in the worst case [85]. This result is unconditional in the sense that it does not depend on any complexity theory assumption. Specifically, it is independent of the famous assumption of $\mathbf{P} \neq \mathbf{NP}$. The authors set forth the proof of hardness by showing that any deterministic algorithm would need an exponential number of queries to the value oracle to solve these two problems.

For the case of additive utilities, the Minimum Envy IRA problem is **NP**-complete since there is a reduction from the Set Partition[6] problem, which is known to be **NP**-complete.

[6]Set Partition problem is the problem of deciding whether a given subset can be partitioned into two subsets of equal sum.

Also, it is shown in [85] that no polynomial time approximation algorithms exist for the problem unless $\mathbf{P} = \mathbf{NP}$. Indeed, using techniques similar to the inapproximability result of the Subset Sum Difference problem, the authors show that for any constant c, no polynomial time algorithm can approximate the Minimum Envy IRA problem within a factor of 2^{m^c}. The Minimum Envy-ratio IRA problem appears to be more tractable on the other hand. In the same paper, Lipton et al. provide a *Polynomial Time Approximation Scheme (PTAS)* for the problem and show that the same scheme serves as a *Fully Polynomial Time Approximation Scheme (FPTAS)* if the number of players is a constant.

Truthful Mechanisms and Envy-freeness

In the past decade, many computer scientists especially algorithmic game theorists, have paid heed to the area of truthful mechanism design, the task of developing algorithms for a class of *private information games* which are robust to untruthfulness on the part of the players competing in the game. The class of private information games refers to non-cooperative settings in which some players, making use of the private piece of knowledge that they possess, independently compete over limited resources. In so doing, they may lie about their private information in an attempt to manipulate the game to their personal advantage. The goal of the algorithm designer is to set up the structure of the game in such a way that it motivates the players to reveal their true private information.

Indivisible resource allocation problems have enjoyed game-theoretic treatments. One good example of such treatments is a mechanism design approach taken toward envy-free allocations of indivisible goods, or more technically, the problems of Truthful Envy-free IRA and Truthful Minimum Envy IRA. In these two settings, there exists no single entity (such as the oracle in minimum envy and minimum envy-ratio problems of Section 1.4.3) that keeps a record of players' valuations. Instead, queries should be directed to each player, in response to which they may lie to increase their benefit. The Truthful Envy-free IRA problem is defined by the following objective and utility functions:

- **Utility function:** monotone or additive, private
- **Objective function:** Among all envy-free allocations, return the one in which all players can maximize their utility functions only by revealing their true valuations

The Truthful Minimum Envy IRA problem is defined by:

- **Utility function:** monotone or additive, private
- **Objective function:** Among all allocations with minimum envy, return the one in which all players can maximize their utility functions only by revealing their true valuations

Lipton et al. prove that the answer sets to both Truthful Envy-free IRA and Truthful Minimum Envy IRA problems are empty, meaning that no envy-free or envy-minimizing mechanism can be truthful [85]. According to them, his impossibility result holds even for a more restricted case of additive utility functions.

Max-Min Allocation of Indivisible Resources

Perhaps the most well-studied concept of fairness in the context of indivisible resources is the notion of Max-Min allocations. The research on this topic started back in early 1980's. The problem of Max-Min allocation of indivisible resources was studied under different names in the earlier stages of its life. The first papers on the topic considered special, usually more tractable, cases under the name of "scheduling jobs on parallel machines" [7, 29, 32, 104, 107], which referred mostly to the cases of related or identical parallel machines (see Section 1.3.1 for a discussion on a general framework for scheduling problems). We will discuss these in Section 1.4.3.

The most general case of the problem was proposed in [85] as a plausible direction for future research, and it was first studied by Bezáková and Dani in [13]. The utility functions are assumed to be additive, therefore for any collection of items C and an arbitrary player $i \in \mathcal{P}$, we have that $f_i(C) = \sum_{j \in C} v_{ij}$, where $v_{ij} = f_i(\{j\})$ is a non-negative real number indicating the value item j has to player i. The fairness is imposed by changing the objective to split the set of items $\mathcal{R}$ into an allocation $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m)$ in such a way that $\min_{i \in \mathcal{P}}\{f_i(\mathcal{A}_i)\}$ is maximized, hence the name Max-Min IRA. Bansal and Sviridenko [10] studied the problem under the name of Santa Claus problem. The name comes from the following restatement of the problem: Santa Claus wants to distribute n presents between m kids. Every present j has some arbitrary value v_{ij} to kid i. The objective is to distribute the presents among the kids in such a way that the happiness of the least happy kid is as high as possible. In other words, Santa's goal is to maximize $\min_{i=1,2,\ldots,m}\{\sum_{j \in \mathcal{A}_i} v_{ij}\}$. Based on the constraints on the values of the items, different sub-variants of the problem can be distinguished. The following cases have been considered in the literature.

Maximizing the Minimum Completion Time on Parallel Machines

Suppose we have a system composed of some machines (e.g. engines) which can only work, or be *alive* if all the machines are alive. Also, suppose that each of these machines requires some resource (e.g. fuel) to function (stay alive). Then, one can ask how to allocate these resources to the machines in such a way that the system stays alive for as long as possible. Such problems have been the motivation for a class of scheduling problems known as Maximizing the Minimum Completion Time, which can be performed on a set of *identical* machines, or some parallel machines with different *speeds* (or sometimes referred to as *related*) machines. Based on the general framework of Graham et al. (See Section 1.3.1),

the problem of identical machines may be represented as $P\,||C_{min}$. C_{min} indicates the minimum completion time and the case of different speeds is denoted by $Q\,||\,C_{min}$.

The case of identical machines is when all players place the same value on each item, or in the context of job scheduling, every job has the same processing time on all the machines. In this case, the load of a machine $i \in \mathcal{M}$ is defined as $\ell_i = \sum_{j \in \mathcal{A}_i} p_j$, where as mentioned before, $\mathcal{A}_i$ is the set of jobs allocated to machine i, and p_j is the processing time of job j. The goal is to maximize the load on the least heavily loaded machine. More precisely, the problem of Maximizing the Minimum Completion Time on Identical Machines is defined as follows:

- **Utility function:** additive, $v_{ij} = p_j$ (processing time of job j)
- **Objective function:** find an allocation $\mathcal{A}$ that maximizes $\min_{i \in \mathcal{M}} \sum_{j \in \mathcal{A}_i} p_j$

The problem of Maximizing the Minimum Completion Time on [Related] Parallel Machines refers to the case where every machine i has a speed s_i, and therefore, the load is defined as $\ell_i = (\sum_{j \in \mathcal{A}_i} p_j)/s_i$:

- **Utility function:** additive, $v_{ij} = p_j$ (processing time of job j)
- **Objective function:** find an allocation $\mathcal{A}$ that maximizes $\min_{i \in \mathcal{M}} \frac{\sum_{j \in \mathcal{A}_i} p_j}{s_i}$

Both problems are known to be **NP**-complete [46]. In the following, we consider different solution methods proposed in the literature toward this problem.

Approximations for $P\,||\,C_{max}$ and $Q\,||\,C_{max}$. Deuermeyer et al. consider an LPT-heuristic (Largest Processing Time first) for the former problem [32]. They show that ordering the jobs in a non-increasing order and assigning each job to the least loaded machine at the moment yields an approximation factor of no worse than 4/3 for the case of identical machines. In [29], the approximation guarantee is further refined to $(3m-1)/(4m-2)$. Woeginger provides a PTAS for the problem of Maximizing the Minimum Completion Time on Identical Machines [107]. Furthermore, Alon et al. consider a more general version of the problem in [2]. While the machines are still identical in their model, the utility functions on them are allowed to be a more general class of convex functions. The machines are identical in the sense that they all use the same function to evaluate each batch of jobs assigned to them. Then, they identify conditions for the utility functions under which a PTAS for the problem of minimizing makespan exists. Their approach uses a classic result of Lenstra [82]. Lenstra shows that any integer linear programming (ILP) with a fixed number of variables can be solved in polynomial time. Alon et al. manage to transform the problem into an ILP formulation with a fixed number of variables, although their running time depends super-exponentially ($(1/\varepsilon^2)^{(1/\varepsilon^2)} + \mathcal{O}(n)$ where n is the number of jobs) on the inverse of the error parameter ε. Later, the running time is improved to

$2^{\mathcal{O}\left(1/\varepsilon^2+\log^3(1/\varepsilon)\right)}+n^{\mathcal{O}(1)}$ by Jansen [68]. The approximation scheme of Jansen is proposed for the more general case of $Q\,||\,C_{max}$ problem which includes $P\,||\,C_{max}$ as a special case. Chen, Jansen, and Zhang show that this running time is very close to optimal by showing a lower bound of $2^{\mathcal{O}(1/\varepsilon)}+n^{\mathcal{O}(1)}$ [24]. For their proof, they use the following widely accepted hypothesis by Impagliazzo et .al:

Definition 8 (Exponential Time Hypothesis (ETH))**.** *[66]: There is a positive real δ such that* 3-SAT *with n variables and m clauses cannot be solved in time $2^{\delta n}(n+m)^{\mathcal{O}(1)}$.*

Then, they show that a $2^{\mathcal{O}\left((1/\varepsilon)^{1-\delta}\right)}+n^{\mathcal{O}(1)}$ time PTAS for $P\,||\,C_{max}$ for any $\delta>0$ implies that ETH fails. Note that ETH, if true, implies the other widely-believed hypothesis that $\mathbf{P}\neq\mathbf{NP}$.

Approximate and exact solutions for $P_m\,||\,C_{max}$ and $Q_m\,||\,C_{max}$. In another research direction, people have looked at cases where m, the number of machines, is not a part of the input, and therefore is treated as a constant. It is particularly interesting to examine how the quality of approximations and the running time of algorithms improve under this simplification. For this case, Horowitz and Sahni propose an FPTAS for the case of identical machines ($P\,||\,C_{max}$) with the running time of $\mathcal{O}\left(nm(nm/\varepsilon)^{m-1}\right)$ where ε is the error parameter [64]. Lenstra, Shmoys, and Tardos improve this result after more than a decade [83]. They give a PTAS with the running time of $(n+1)^{m/\varepsilon}poly(|\,\mathcal{I}\,|)$ and a space complexity polynomial in $1/\varepsilon$, m, and $|\,\mathcal{I}\,|$, where $|\,\mathcal{I}\,|$ is the size of the input[7]. In [70], Jansen and Prokolab improve the running time to $n(m/\varepsilon)^{\mathcal{O}(m)}$, settling the question of finding an FPTAS for the problem raised by Lenstra et al. in [83]. A similar result is also shown for a more general case of $R_m\,||\,C_{max}$. Jansen and Mastrolilli further improve the time complexity of the FPTAS to $\mathcal{O}\left(n\right)+(1/\varepsilon)^{\mathcal{O}(m)}$ for sufficiently large values of m $(m>1/\varepsilon)$[8] [69]. Chen, Jansen, and Zhang prove that any $(1/\varepsilon)^{\mathcal{O}\left(m^{1-\delta}\right)}+n^{\mathcal{O}(1)}$ time FPTAS for the problem for $\delta>0$ implies that ETH fails [24].

In a recent trend of research, some of the scientists in the field have investigated more time-efficient exact solutions of variations of the problem. One of the problems that has enjoyed such treatments in $P_m\,||\,C_{max}$, which is **NP**-hard to solve optimally. Nevertheless, finding "relatively fast" (compared to naive approaches) exponential algorithms that can solve the problem optimally is far from trivial. As an instance, mention the work of O'Neil and Kerlin who provide an exact solution to $P_m\,||\,C_{max}$ with the running time of $2^{\mathcal{O}\left(\sqrt{m}|\,\mathcal{I}\,|+m\log|\,\mathcal{I}\,|\right)}$ [91] and a recent result by Lenté et al. who give a $2^{n/2}$ algorithm in time complexity for the case of $P_2\,||\,C_{max}$ (two identical parallel machines), and a $3^{n/2}$ time algorithm for $P_3\,||\,C_{max}$ [84]. Chen et al. provide lower bounds for this model as well [24]. They show that for the case where the processing times are bounded by $\mathcal{O}\left(n\right)$, an exact algorithm of running time $2^{\mathcal{O}\left(n^{1-\delta}\right)}$ for any $\delta>0$ implies that ETH fails.

[7]The algorithm of Lenstra, Shmoys, and Tardos is in fact proposed for the broader class of unrelated parallel machines or $R_m\,||\,C_{max}$.

[8]This result is also proposed for the more general case of $R_m\,||\,C_{max}$.

Online $P\,|\,|\,C_{max}$ and $Q\,|\,|\,C_{max}$. Azar and Epstein consider the latter problem in the *on-line* setting in [7]. They assume that a sequence of jobs is dynamically scheduled on a set of m identical machines, i.e., each job has to be allocated to a machine immediately when considered, and this allocation cannot be modified at any future stage. In this scenario, the entire input is not available to the algorithm at any given time. Instead, it is fed to the algorithm as a read-once sequence. Azar and Epstein improve a previous result by Woeginger who showed a greedy algorithm is an m-competitive algorithm for the on-line version of Maximizing the Minimum Completion Time on [Related] Parallel Machines [107]. The *competitive ratio* of an on-line algorithm is the ratio between the worst (expected) performance of the algorithm for arbitrary input sequences compared to the optimal off-line solution (if all the input was given to the algorithm at once). They provide a randomized on-line algorithm which is $\mathcal{O}\left(\sqrt{m}\log m\right)$-competitive, and also manage to show that any randomized on-line scheduler is at least $\Omega\left(\sqrt{m}\right)$-competitive. For the case of identical machines, they give a $\mathcal{O}\left(\log F\right)$-competitive randomized on-line algorithm, where F is the ratio between the longest and shortest processing times of the jobs. In the case that this ratio is a polynomial in the number of machines m, this algorithm readily gives a $\mathcal{O}\left(\log m\right)$-competitive algorithm, which asymptotically matches the lower bound of $\Omega\left(\log m\right)$-competitiveness that the authors prove for any randomized on-line scheduler on identical machines. Finally, in a more recent work, Tan et al. have looked at the on-line setting again, only this time under certain assumptions [104]. They consider the problem of Maximizing the Minimum Completion Time on [Related] Parallel Machines for a simplified setting of only two machines, one with a speed of $s_1 = 1$, and the other one with a speed of $s_2 = s \geqslant 1$. In their model, the scheduler is confronted with a sequence of independent jobs, $p_1, p_2, \ldots, p_n$ in an on-line fashion. The characteristic that distinguishes their model from the previous work is the assumption of *ordinal data*, that is, the processing times of the jobs are not known, but the sorted order of the jobs is known in advance. The authors of [104] provide a lower bound on the competitive ratio of any algorithm for this problem in the format of a piecewise function of the speed ratio s. Then, they present an optimal parametric algorithm for any given s. More precisely, they prove the following lower bound which as the authors show, is a well-defined function.

$$c(s) = \begin{cases} \frac{2(2k-1)s}{ks+2(k-1)}, & a_{k-1} \leqslant s < b_k, \quad k \geqslant 2 \\ \frac{2k+1}{k+1}, & b_k \leqslant s < a_k, \qquad k \geqslant 2 \\ 2, & s \geqslant 2 \end{cases}$$

where $a_k = \frac{2k}{k+1}$ and $b_k = \frac{2(k-1)(2k+1)}{2k^2+k-2}$. They also provide an algorithm that matches these bounds.

General Santa Claus Problem

- **Utility functions:** additive, item values are non-negative
- **Objective function:** find an allocation $\mathcal{A}$ that maximizes $\min_{i\in\mathcal{P}}\{\sum_{j\in\mathcal{A}_i} v_{ij}\}$

Although very similar to the problem of Make-span Minimization on Unrelated Machines in Section 1.4.3, this problem is completely different in essence [5, 13], which means using the techniques for minimizing the makespan would not yield any nontrivial result for the case of Max-Min Fair Allocation problem. General Santa Claus problem received its first treatment by Bezákova and Dani [13]. The authors mention a reduction from Subset Sum problem to General Santa Claus problem for the special case of two players, pointing out **NP**-hardness of the problem. They also use another reduction from 3D-Matching problem to show that there exists no polynomial-time α-approximation algorithms for $\alpha > 1/2$. Then they use the same integer linear programming formulation as [83], called the *assignment LP*. Here, $x_{i,j}$ is the indicator variable that means player i gets item j.

$$\max \quad \omega \tag{1.4.1}$$

$$\text{s.t.} \quad \sum_{i\in\mathcal{P}} x_{i,j} = 1 \qquad \forall j \in \mathcal{R} \tag{1.4.2}$$

$$\sum_{j\in\mathcal{R}} v_{ij}\cdot x_{i,j} \geqslant \omega \qquad \forall i \in \mathcal{P} \tag{1.4.3}$$

$$x_{i,j} \in \{0,1\} \qquad \forall j \in \mathcal{R},\ \forall i \in \mathcal{P} \tag{1.4.4}$$

Then, they relax the integrality constraint of Equation (1.4.4) to $0 \leqslant x_{i,j} \leqslant 1$, and use the same rounding technique as in [83] to get an additive factor loss of $\max_{i,j} v_{ij}$ in approximation quality, which can be arbitrarily bad.

Bansal and Sviridenko revisit the problem using a stronger LP formulation [10]. They use machine scheduling terminology, however (following the footsteps of earlier works on the topic such as [107]), where kids correspond to machines and presents correspond to jobs to be scheduled on these machines. The value of a present to a kid corresponds to the processing time of a job on a machine in this new setting. The goal is to schedule jobs on the machines in such a way that the minimum load is maximized. As before, in the setting of scheduling we use the set of jobs, denoted by $\mathcal{J}$ to mean the set of items $\mathcal{R}$, and the set of machines, denoted by $\mathcal{M}$ interchangeably with the set of players, $\mathcal{P}$. Also, instead of values v_{ij}, we use p_{ij} to denote the processing time of a job j on a machine i. In their new LP formulation, Bansal and Sviridenko look at all possible subsets of jobs known as *configurations*. Therefore, the LP they propose is named *configuration LP* and is written in the format of an exponentially large (in terms of the number of constraints) feasibility linear

programming. To explain further, assume we are given T, the value of the objective function, i.e., the minimum machine load that every machine should have after allocation. We can use binary search over the range of T to get as close as desired to our target objective. Consider a configuration C which is a subset of jobs (items). Let the size of the configuration on machine (player) i, denoted by $size(C, i)$, be the sum of the processing times (values) of all the jobs (items) in C. Now, we define the notion of *valid* configurations.

Definition 9 (Valid Configurations)**.** *A configuration C is valid for machine i under a given objective value T if $size(C,\, i) \geqslant T$.*

The set of valid configurations for machine i under objective value T is denoted by $\mathcal{C}(i,\, T)$. There is a variable $x_{i,\,C}$ for every valid configuration $C \in \mathcal{C}(i,\, T)$ on machine i (note that it is not a binary variable, and one can think of it as a real-valued version of an indicator variable). The number of such variables is potentially exponential. The configuration LP will be defined as:

$$\min \quad 0 \tag{1.4.5}$$

$$\text{s.t.} \quad \sum_{C \in \mathcal{C}(i,\, T)} x_{i,\, C} \geqslant 1 \qquad \forall i \in \mathcal{M} \text{ (or } \mathcal{P}) \tag{1.4.6}$$

$$\sum_{C \ni j} \sum_{i} x_{i,\, C} \leqslant 1 \qquad \forall j \in \mathcal{J} \text{ (or } \mathcal{R}) \tag{1.4.7}$$

$$x_{i,\, C} \geqslant 0 \qquad \forall i \in \mathcal{M} \text{ (or } \mathcal{P}) \text{ , } \forall C \in \mathcal{C}(i,\, T) \tag{1.4.8}$$

The dual of this linear programming is often used in the analyses [94], as we shall explain in a few paragraphs. We provide the dual of the configuration LP below. Assuming dual variables y_i for $1 \leqslant i \leqslant m$ correspond to Equation (1.4.6) and variables z_j for $1 \leqslant j \leqslant n$ correspond to Equation (1.4.7), we write the dual as:

$$\max \quad \sum_{i=1}^{m} y_i - \sum_{j=1}^{n} z_j \tag{1.4.9}$$

$$\text{s.t.} \quad \sum_{j \in C} z_j \geqslant y_i \qquad \forall i \in \mathcal{M} \text{ (or } \mathcal{P}) \text{ ,} \forall C \in \mathcal{C}(i,\, T) \tag{1.4.10}$$

$$y_i, z_j \geqslant 0 \qquad \forall i \in \mathcal{M} \text{ (or } \mathcal{P}) \text{ , } \forall j \in \mathcal{J} \text{ (or } \mathcal{R}) \tag{1.4.11}$$

The authors of [10] first show that the integrality gap of LP formulation of [13] is $\Omega\,(m)$. The result holds even when all jobs have the same processing times on all machines. Then, they turn their attention to the integrality gap of the stronger configuration LP.

Theorem 1.4.3. *[10] The configuration LP of Equation* (1.4.5)*–Equation* (1.4.8) *has an integrality gap of $\Omega\,(\sqrt{m})$ for arbitrary processing times p_{ij}.*

To achieve this, Bansal and Sviridenko demonstrate a class of problem instances in which m, the number of players or machines, is set to be $\mathcal{O}(k^2)$ for some integer k. They show that the optimal fractional solution of the configuration LP is $\Omega(k)$ while the optimal integral solution is 1. A solution of value OPT/m is proposed in [10] as well, where OPT is the optimal value of T, the solution to the General Santa Claus problem. The authors also make use of the fractional solution of configuration LP for the case of Restricted Santa Claus problem, which will be explained briefly in the next section. To show both approximation algorithms for the general and restricted case take polynomial time, the configuration LP must be solvable in polynomial time in the first place. To that end, Bansal and Sviridenko use the *separation problem* of the dual. They first show that the separation problem to the dual is a regular Minimum Knapsack problem, which can be solved to any desired precision ε in polynomial time. We bring the definition of the separation problem here for the sake of completeness.

Definition 10 (The Separation Problem)**.** *[108] The* separation problem *associated with a combinatorial optimization problem* $\max\{cx : x \in X \subseteq \mathbb{R}^n\}$ *is the problem: given* $x^* \in \mathbb{R}^n$*, is* $x^* \in conv(X)$ [9]*? If not, find an inequality* $\pi x \leqslant \pi_0$ *satisfied by all points in* X*, but violated by the point* x^**.*

Since the separation problem for the dual can be solved to any arbitrary precision in polynomial time, the ellipsoid method can solve the dual in polynomial time as well. Therefore, one can find a feasible solution to the primal to any desired accuracy in polynomial time.

Subsequently, Asadpour and Saberi provide an iterative rounding scheme in [6] to get an objective value no worse than $\mathcal{O}\left(\frac{OPT}{\sqrt{m}(\log^3 m)}\right)$, which counts as the first non-trivial approximation algorithm for the General Santa Claus problem. They first split the items into two sets: the ones that have value more than $\frac{T}{\sqrt{m}\log^3 m}$ are called *big* items, and the rest are called *small* items. Then, they carry out their iterative rounding based on a bipartite graph matching. Their rounding method ensures each player either receives a big item or is satisfied with a bundle of small items that values a constant fraction of a big item's, hence the $\mathcal{O}\left(\frac{OPT}{\sqrt{m}(\log^3 m)}\right)$ approximation guarantee. There are polylogarithmic approximation factor algorithms with quasi-polynomial running times independently by Bateni et al. [11] and Chakrabarty et al. [21], but since the techniques used in these two papers are mostly devised for special cases of the Santa Claus problem, we shall say more on these results in Section 1.4.3. Finally, Saha and Srinivasan in [98] get a better approximation factor of $\mathcal{O}\left(\sqrt{\frac{\log\log m}{m\log m}}\right)$ with yet another rounding technique, which is very close to the integrality gap of the configuration LP (see Theorem 1.4.3).

[9] $conv(X)$ is the convex hull of X

Restricted Santa Claus Problem

- **Utility functions:** additive, item values $v_{ij} \in \{0, v_j\}$
- **Objective function:** find an allocation $\mathcal{A}$ that maximizes $\min_{i \in \mathcal{P}}\{\sum_{j \in \mathcal{A}_i} v_j\}$

In the restricted case, for every item $i \in \mathcal{R}$ and every player $j \in \mathcal{P}$, it is assumed that $v_{ij} \in \{0, v_i\}$, where v_i is a positive real number. In other words, every item i has either a fixed value v_i to all those players who are interested in that item, or a value of 0 to those players not interested in i. The same 1/2-inapproximability result of [13] holds for this case as well. Also, the rounding technique of [10] can be used for a configuration LP in the restricted case. In this setting, Bansal and Sviridenko provide an $\mathcal{O}\left(\frac{\log\log\log m}{\log\log m}\right)$ factor approximation algorithm [10]. Feige proves that the integrality gap of configuration LP is a constant, although he does not provide a polynomial time algorithm to provide a constant factor approximation of the optimal solution [37]. What he has proposed is, in fact, an estimation algorithm. Feige himself points out the distinction between an approximation algorithm and an estimation algorithm as follows:

> "*In some cases the design of approximation algorithms includes a nonconstructive component. As a result, the algorithms become estimation algorithms rather than approximation algorithms: they allow one to estimate the value of the optimal solution, without actually producing a solution whose value is close to optimal.*" [38]

We should mention that hardness results on approximation algorithms remain valid even for estimation algorithms. More to the point, estimating the optimal value of General Santa Claus and Restricted Santa Claus problems within a factor of $\frac{1}{2}$ cannot be done in polynomial time unless $\mathbf{P} = \mathbf{NP}$. Recently, Haeupler et al. have given a constructive version of Feige's proof using the constructive proof of Lovász Local Lemma [58]. It counts as the first constant factor approximation algorithm for Restricted Santa Claus problem, although they do not provide any specific constant in their algorithm.

Asadpour et al. use solution concepts from 3D-Matching problem alongside configuration LP [5]. They first use some sufficient condition for the existence of matchings in hypergraph by Haxell [61] to prove a conjecture proposed in [10] by Bansal and Sviridenko, which in turn proves that the integrality gap of configuration LP is no worse than $\frac{1}{3}$. Then, they propose an algorithm that achieves a 1/4-approximation solution to the Restricted Santa Claus problem in exponential time. Their algorithm works around the notions of big and small items as in [10] and [6]. After categorizing the items into these two sets, they build a hypergraph-based on the solution to the configuration LP relaxation. The hypergraph is used to find a hypergraph matching which is the equivalent of an allocation $\mathcal{A}$ satisfying the condition $\min_{i \in \mathcal{P}}\{\sum_{j \in \mathcal{A}_i} v_j\} \geqslant \frac{T}{4}$, where T is the optimal value, i.e., the

objective value of the allocation that maximizes the benefit of the least lucky player. To find the matching, they perform a local search on an *alternating tree*. Edmonds first introduced the notion of alternating trees in the algorithm he proposed for maximum matching in general graphs, known as the *blossom algorithm* [36]. Their local search requires exponential time. Recently, Polacek and Svensson have managed to show that for the local search, one just needs to consider the alternating tree to a limited height, hence cutting the running time from exponential to quasi-polynomial time of $m^{\mathcal{O}(\log m)}$.

Special Cases

The following case of the Max-Min problem, called $(0, 1, U)$-MAX-MIN$^+$, has been shown to be as hard as the GENERAL SANTA CLAUS problem [74]:

- **Utility functions:** additive, item values $v_{ij} \in \{0, 1, U\}$ for some $U > 1$
- **Objective function:** find an allocation $\mathcal{A}$ that maximizes $\min_{i \in \mathcal{P}}\{\sum_{j \in \mathcal{A}_i} v_{ij}\}$

The authors of [74] provide a trade off between running time and approximation guarantee through an algorithm that, for any given $\alpha < m/2$, finds an allocation of value greater than or equal to $\alpha \cdot OPT/m$ in time $n^{\mathcal{O}(1)}m^{\mathcal{O}(\alpha)}$. Prior to [74], Golvin had shown a $\mathcal{O}(\sqrt{m})$-approximation algorithm for this variant [53].

In [11], Bateni et al. consider a case where every item has a positive value for a limited number of players:

- **Utility functions:** additive, any item i has value $v_{ij} > 0$ for some D players, 0 for the rest
- **Objective function:** find an allocation $\mathcal{A}$ that maximizes $\min_{i \in \mathcal{P}}\{\sum_{j \in \mathcal{A}_i} v_{ij}\}$

Definition 11 (Bipartite Graph Representation)**.** *A bipartite graph representation of an instance of the* SANTA CLAUS *problem in general is a bipartite graph $H = (\mathcal{P}, \mathcal{R}, E)$ in which vertices in $\mathcal{P}$ correspond to players, vertices in $\mathcal{R}$ correspond to resources or items, and there is an edge $e = (i, j) \in E$ for $i \in \mathcal{P}$ and $j \in \mathcal{R}$ if the item j has positive value to player j.*

This variant of the problem is called BOUNDED DEGREE MAX-MIN IRA. In the bipartite graph representation of the problem, every item has a bounded degree of D, hence the name. The authors show that any instance of the GENERAL SANTA CLAUS problem can be cast as another instance with bounded degree $D = 3$. For $D = 2$, they further divide the problem into two sub-classes: *symmetric* and *asymmetric*. In the symmetric case, every item has the same value for both players to whom it is connected. In the asymmetric case, it can have different values for the two players. Bateni et al. show that the case of bounded degree $D = 2$ cannot be approximated within a factor larger than $\frac{1}{2}$ via a reduction from the 3SAT problem, even for the symmetric sub-class [11]. They also provide a 1/4-approximation

algorithm for the asymmetric case and give a simpler LP formulation for the general case (General Santa Claus) called *M-LP*:

$$\begin{aligned}
\min \quad & 0 && && (1.4.12)\\
\text{s.t.} \quad & \sum_{i\in\mathcal{P}} x_{i,j} \leqslant 1 && \forall j \in \mathcal{R} && (1.4.13)\\
& \sum_{j\in\mathcal{R}\,:\,v_{ij}=T} x_{i,j} + z_i \geqslant 1 && \forall i \in \mathcal{P} && (1.4.14)\\
& \sum_{j\in\mathcal{R}\,:\,v_{ij}<T} v_{ij}\cdot x_{i,j} \geqslant z_i \cdot T && \forall i \in \mathcal{P} && (1.4.15)\\
& x_{i,j} \leqslant z_i && \forall i \in \mathcal{P}, j \in \mathcal{R} \,:\, v_{ij} < T && (1.4.16)\\
& x_{i,j}, z_i \geqslant 0 && \forall i \in \mathcal{P}, j \in \mathcal{R} && (1.4.17)
\end{aligned}$$

in which, $x_{i,j}$ indicates the fraction of item j assigned to player i Also, z_i is called the *small usage* of player i and indicates how much small items contribute to the utility of player i. Finally, T is an estimate or guess of the optimal objective value found via binary search. An interesting feature of this formulation is that, in contrast to the configuration LP, it has a polynomial number of variables and constraints. Building upon this new formulation, the authors of [11] devise a polylogarithmic approximation algorithm for the general problem that runs in quasi-polynomial time. Chakrabarty et al. have also achieved the same results. [21].

In [86], a sub-class of the Restricted Santa Claus is considered:

- **Utility functions:** additive, item values $v_{ij} \in \{0, v_j\}$, the bipartite graph representation forms an inclusion-free convex bipartite graph
- **Objective function:** find an allocation $\mathcal{A}$ that maximizes $\min_{i\in\mathcal{P}}\{\sum_{j\in\mathcal{A}_i} v_j\}$

A bipartite graph representation $H = (\mathcal{P}, \mathcal{R}, E)$ (or any bipartite graph for that matter) is said to be *convex bipartite*, or *ordered* if there exists an ordering of the vertices in $\mathcal{P}$ such that $N_H(i)$, the neighbourhood of every vertex $i \in \mathcal{P}$ forms an interval. By an interval we mean a continuous sequence of the vertice in cR. H is said to be *inclusion-free convex bipartite* if it is convex bipartite, and no neighbourhood $N_H(i)$ falls completely within another one. Authors of [86] provide a 1/2-approximation algorithm for this sub-class.

Truthfulness and Max-Min

A game theoretic approach to Max-Min IRA has been taken in [13]. Similar to Section 1.4.3, we assume that the utility functions of the players are private, meaning the allocation algorithm cannot access the private value a player has for an item, and it must ask the player

about the valuation. The players in return may choose to lie in order to manipulate the algorithm into allocating them a more valuable chunk of items. This variant of the problem is defined as follows:

- **Utility functions:** additive, item values are non-negative and private
- **Objective function:** find an allocation $\mathcal{A}$ that maximizes $\min_{i\in\mathcal{P}}\{\sum_{j\in\mathcal{A}_i} v_{ij}\}$, while each player can maximize her benefit by revealing her true valuation

Bezákova and Dani show that for two players, the expected minimum of a randomized *divide-and-choose* mechanism is at least $OPT/2$ [13]. To complement this result, they also prove that no truthful mechanism can find the optimum allocation. For more on divide-and-choose strategies, consult [16].

Min-Max Allocation of Indivisible Resources

For long, Min-Max scenarios have mostly been considered in the context of scheduling problems for obvious reasons: some jobs are scheduled on a set of machines which run the jobs in parallel. The completion time of the entire batch of jobs is the maximum completion time or make-span among all the machines. Thereby, one direction to look at is minimizing the maximum makespan. As usual with the job scheduling settings, we switch to our alternative notation. In the language of the "three-field" notation of Graham et al. (see Section 1.3.1), we will consider the following variants of the $R||C_{\max}$ problem:

General $R||C_{\max}$

The General $R||C_{\max}$ problem was considered in [83] by Lenstra, Tardos, and Shmoys, in one of the most cited papers in the field. The problem is defined in the following way:

- **Utility functions:** additive, processing times are non-negative
- **Objective function:** find an allocation $\mathcal{A}$ that minimizes $\max_{i\in\mathcal{M}}\{\sum_{j\in\mathcal{A}_i} p_{ij}\}$

In [83], the authors use the famous assignment LP, which we mentioned in Section 1.4.3 (Equation (1.4.1)–Equation (1.4.4)). Their main result is a 2-approximation algorithm for the problem They first solve the linear programming relaxation of the assignment LP in polynomial time to get a fractional assignment of jobs to machines. Then, they provide a rounding scheme for the extreme points of this LP to integral solutions. After their rounding, the feasible integral solution obtained is within a factor of 2 of OPT, where OPT is the optimal solution to the original (integral) General $R||C_{\max}$ problem. To complement this result, they also show that the problem cannot be approximated to a factor better than $\frac{3}{2}$ unless $\mathbf{P} = \mathbf{NP}$ via a reduction from 3D-Matching problem. Despite being over twenty

years old now, these results are the best approximation algorithm and hardness result ever known. Thus, a gap of $\frac{1}{2}$ still stands between the known boundaries of the problem today.

For the case of fixed number of machines, or $R_m\,||\,C_{max}$, all the approximation results mentioned for $P_m\,||\,C_{max}$ (see Section 1.4.3) apply to this broader model as well. As a matter of fact, all the mentioned algorithms were designed and proposed with $R_m\,||\,C_{max}$ problem in mind, but they also apply to the special cases of $P_m\,||\,C_{max}$ and $Q_m\,||\,C_{max}$.

Restricted $R\,||\,C_{\max}$

In the same manner that a restricted version of Santa Claus problem was defined in Section 1.4.3, we can also define the Restricted $R\,||\,C_{\max}$ problem. We assume that jobs can only be processed on a subset of the machines in a reasonable amount of time, and have infinite running time on the rest:

- **Utility functions:** additive, processing times $p_{ij} \in \{\infty, p_j\}$
- **Objective function:** find an allocation $\mathcal{A}$ that minimizes $\max_{i\in\mathcal{M}}\{\sum_{j\in\mathcal{A}_i} p_j\}$

The previously known inapproximability factor of $\frac{3}{2}$ for the general case also holds for the restricted version, unless $\mathbf{P} = \mathbf{NP}$. Svensson has considered this variant of the problem [102]. The author uses the configuration LP (Equation (1.4.5)–Equation (1.4.8)) which has already been used for the Santa Claus problem to estimate the optimal completion time within a factor of $33/17 + \varepsilon \approx 1.9412 + \varepsilon$ for small values of ε. We note that this algorithm is an estimation algorithm but not an approximation algorithm, the reason being that Svensson's algorithm heavily relies on a local search procedure that is not known to converge in polynomial time.

Special Cases

Most of the special cases of the $R\,||\,C_{\max}$ problem put certain restrictions on the way the jobs can be assigned to the machines. For the sake of convenience, we define the set of *eligible* machines for a job as follows:

Definition 12 (The Set of Eligible Machines)**.** *For any arbitrary job $j \in \mathcal{J}$, the set of* eligible *machines for j denoted by M_j ($M_j \subseteq \mathcal{M}$) is a subset of machines on which j has finite running time (or can be processed on these machines).*

This definition comes in handy in the discussion of the special cases regarded in this section. The first one is the case of Parallel Machine Scheduling with Grade-of-Service Eligibility studied by Hwang et al. [65]. In this case, the customers make requests for the machines, and the objective is, as with other variants of $R\,||\,C_{\max}$ problem, to minimize the makespan. The property that is unique to this model is that the customers are of different values to the service provider. For instance, some are considered to be silver,

gold, or platinum members. Therefore, the service provider differentiates in the policy of service for each of these classes of customers. The way Hwang et al. propose to implement this differentiation is to assign a GoS (Grade-of-Service) value to both machines and jobs, and then allow each job to be processed on only those machines that have a GoS less than or equal to that of the job. The problem can be formally defined as:

- **Utility functions:** additive, for each processing time p_{ij} we have $p_{ij} \in \{\infty, p_j\}$ such that for every two jobs $j, k \in \mathcal{J}$, either $M_j \subseteq M_k$ or $M_k \subseteq M_j$
- **Objective function:** find an allocation $\mathcal{A}$ that minimizes $\max_{i \in \mathcal{M}}\{\sum_{j \in \mathcal{A}_i} p_j\}$

This case also arises in situations where each job has a memory requirement, and every machine has a certain amount of memory available to offer which is inherently different from other processors. In this case too, a *total ordering* of the jobs is formed in the sense that each pair of jobs is comparable via the inclusion criterion of their corresponding set of eligible machines. For this case, Hwang et al. provide a polynomial time approximation algorithm with an approximation factor of no worse than $2 - 1/(m-1)$ [65]. Glass and Kellerer consider the same problem and manage to improve the approximation quality to 3/2 [49]. Finally, Ou et al. provide a PTAS for the problem [92].

Glass and Kellerer also considered a slightly different case in which a total ordering of the job does not exist, but the inclusion of the eligible machines is still preserved [49]. In their model, for any two jobs $j, k \in \mathcal{J}$, one of the following holds: i) $M_j \cap M_k = \varnothing$, ii) $M_j \subseteq M_k$, iii) $M_k \subseteq M_j$. This sub-variant is referred to as the Parallel Machine Scheduling with Nested Job Assignments problem. They propose a polynomial time algorithm that approximates the minimum makespan within a factor of $2 - \frac{1}{m}$ and mention the question of finding a polynomial time approximation scheme as an open problem, a question which is addressed in [88] by Munatore, Schwarz, and Woeginger.

Ebenlendr et al. consider a very special case of the Restricted $R\,|\,|\,C_{\max}$ where each job can be scheduled on at most two machines [35]. They call this case the Graph Balancing problem.

- **Utility functions:** additive, for each processing time p_{ij} we have $p_{ij} \in \{\infty, p_j\}$ such that for every job $j \in \mathcal{J}$, $|\,M_j\,| \leqslant 2$
- **Objective function:** find an allocation $\mathcal{A}$ that minimizes $\max_{i \in \mathcal{M}}\{\sum_{j \in \mathcal{A}_i} p_j\}$

Finding an exact solution to Graph Balancing problem is hard as there is a straightforward reduction from Subset Sum problem when there are only two machines in the system. The authors show that this special case is **NP**-hard to approximate within a factor better than $\frac{3}{2}$. Then they provide a 1.75-approximation algorithm.

1.5 Thesis Overview

The main results of the thesis are organized into two parts. Part I discusses a computational solution to variations of the vehicle routing problem. More specifically, Chapter 2 explains our efforts in solving Skill Vehicle Routing Problem through a column generation technique. These efforts lead us to a subproblem, Prize-Collecting Travelling Salesman, which is a variation of the well-known TSP. Chapter 3 demonstrates a branch-and-cut algorithm that can solve PCTSP computationally in an efficient way. This chapter contains our most significant contribution to the vehicle routing problem, and elaborates algorithms that we use for both the SVRP and the PCTSP. Part II of the thesis focus on the classic Scheduling problem. We study particular instances of the problem known as the ordered instances. In Chapter 4, we present a Polynomial Time Approximation Scheme using dynamic programming. We show that, with minor modifications, our algorithm works for various ordered instances. An overview of the thesis is as follows:

Chapters 2 - A Column Generation Approach to Skill Vehicle Routing Problem: In this chapter, we consider a variation of the vehicle routing problems known as the Skill Vehicle Routing Problem (SVRP). We argue why a solution method based on column generation can be useful in solving the problem to optimality. We provide a branch-and-price algorithm for the problem and discuss that the subproblem is the PCTSP. We also show how we can use the dual variables for generating bounds for the SVRP.

Chapter 3 - A Branch-and-Cut Algorithm for Prize Collecting Travelling Salesman Problem [71]: In this chapter, we provide a complete framework for solving the PCTSP based on a branch-and-cut algorithm. At the heart of our framework lies a cutting plane algorithm which uses two classes of cuts for improving the linear programming solution: Generalized Subtour Elimination Constraints (GSECs) and Primitive Comb Inequalities. We provide efficient heuristics to obtain the GSECs for the PCTSP and compare its performance with an optimal separation procedure. Furthermore, we show that a heuristic to separate the primitive comb inequalities for the TSP can be applied to separate the primitive comb inequalities introduced for the PCTSP. We evaluate the effectiveness of these inequalities in reducing the integrality gap for the PCTSP. We also provide a local search heuristic to help the branching algorithm in finding effective bounds. Finally, we introduce branching heuristics that can guide the search on obtaining feasible solutions quickly. We compare the performance of two of the heuristics against each other, and against the generic branch-and-bound algorithm used by CPLEX.

Chapter 4 - PTAS for Ordered Instances of the Resource Allocation Problems [72]: We consider the problem of allocating a set I of m indivisible resources (items) to

a set P of n customers (players) competing for the resources. Each resource $j \in I$ has a same value $v_j > 0$ for a subset of customers interested in j, and zero value for the remaining customers. The utility received by each customer is the sum of the values of the resources allocated to her. The goal is to find a feasible allocation of the resources to the interested customers such that for the Max-Min allocation problem (Min-Max allocation problem) the minimum of the utilities (maximum of the utilities) received by the customers is maximized (minimized). The Max-Min allocation problem is also known as the *Fair Allocation problem*, or the *Santa Claus problem*. The Min-Max allocation problem is the problem of Scheduling on Unrelated Parallel Machines, and is also known as the $R||C_{\max}$ problem. In this chapter, we are interested in instances of the problem that admit a Polynomial Time Approximation Scheme (PTAS). We show that an ordering property on the resources and the customers is important and paves the way for a PTAS. For the Max-Min allocation problem, we start with instances of the problem that can be viewed as a *convex bipartite graph*; a bipartite graph for which there exists an ordering of the resources such that each customer is interested in (has a positive evaluation for) a set of *consecutive* resources. We demonstrate a PTAS for the inclusion-free cases. This class of instances is equivalent to the class of bipartite permutation graphs. For the Min-Max allocation problem, we also obtain a PTAS for inclusion-free instances. These results are not only of theoretical interest but also have practical applications. Finally, we extend our method to cases of $R||C_{\max}$ problem in which hierarchical assignment restrictions exist. We first provide a PTAS based on our dynamic programming technique for instances in which the hierarchical structure is given in a form known as the laminar families. Next, we expand it to a more general case that we call the "extended laminar families" of sets.

Part II

Skill Vehicle Routing Problem

Chapter 2

A Column Generation Approach to Skill Vehicle Routing Problem

2.1 Introduction and Problem Definition

Vehicle Routing Problem (VRP) deals with the question of routing a fleet of vehicles, typically starting from a centralized site called the *depot*, to visit various service locations scattered over a given area. It is one of the most common challenges in operations research [52] with immediate applications to transportation. Transportation cost is a significant portion of the total cost of a product. According to Rodrigue et. al, " it is not uncommon for transport costs to account for 10% of the total cost of a product" [96]. With the growing popularity of online shopping and online ordering, the scale of the transportation problems businesses face grows steadily as well. Manual route selection and dispatching, solely based on human expertise, no longer can keep up with the demand. The need for automatic optimization software is evident. Computer optimization tools have been able to make a significant impact on transportation costs[47]. Consequently, more companies turn to computer-aided route optimization tools each year. Most of the logistics tasks carried out on a daily basis in courier or delivery services are closely tied to one variation of the VRP or the other. The version we study in this thesis is referred to as the *Skill Vehicle Routing Problem* (SVRP), which has applications for utility service providers and courier companies.

In SVRP, the service location, also known as *customers* are spread out on a two-dimensional metric (usually Euclidean) plane. We model the travel distances between the customers via a metric graph $G = (V, E)$, in which $V = S \cup \{r\}$. S is the set of customer sites and r is the depot, where all vehicles start and end their trips. Associated with the edges is a cost function $c : E \to \mathbb{R}_{\geqslant 0}$. The cost function associates a non-negative value c_{uv} to each arc $(u, v) \in E$ proportionate to the distance between the nodes u and v. We assume that $c_{uu} = 0$ for each $u \in V$. In this variation of the VRP, every vehicle in the fleet is associated with a driver. D represents the set of all drivers, and every driver $i \in D$ has

a particular set of skills suitable for a subset of tasks. Every customer's demand requires exactly one of these skills. Therefore, not all drivers can be sent to visit a given customer. The objective is to find routes that start and end at the depot and assign drivers to them so that:

(1) the union of all the nodes visited by the routes covers the set of customers S

(2) the driver assigned to each route has the required skill at each site she should visit

(3) the total distance travelled by all drivers is minimized

2.1.1 Related Work

Column generation has been a prominent solution technique for vehicle routing problems. The first applications of column generation for vehicle routing problems date back to early 1990's. Desrochers et al. use column generation in a Dantzig-Wolfe decomposition to tackle a *Vehicle Routing Problem with Time Windows* (VRPTW) [31]. In this model, the distances between the customers are given in terms of the travel time. Every customer $j \in S$ is associated with a time window $[a_j, b_j]$. The driver should arrive before the time b_j to serve j. Also, if the driver arrives before the time window opens, she has to wait until the time a_j. In the same year, Halse implements the framework based on a Lagrangian decomposition [60]. Lagrangian decomposition is a variable splitting technique that can be considered as a special case of the Lagrangian relaxation. A Lagrangian relaxation algorithm for the problem of VRPTW is due to Kohl and Madsen [76]. A series of papers in the late 1990's improve on the previous decomposition methods and mix them with cutting plane and parallel algorithms. See the papers by Kohl et al. [75], Larsen [78], and Cook and Rich [26] as examples. More recent research on the VRPTW includes combinations of Dantzig-Wolfe decomposition and Lagrangian relaxation algorithm. [20, 41, 67, 97].

Many other variations of the VRP have also benefited from a column generation solution. One major application is for vehicle routing problems with heterogeneous vehicles where vehicles are different in capacity or cost they incur. We refer the readers to the work of Choi and Tcha [25] on the lower bounds of variations of heterogeneous fleet VRP. Also Choi and Tcha [25] and Taillard [103] provided a heuristic based on column generation for the problem. Please consult [30] and [52] for a thorough treatment of the column generation based methods for solving the VRP.

2.1.2 Our Contribution

We provide a solution framework based on the column generation technique. Our method can be used to obtain exact or near-optimal solutions to the SVRP. Through SVRP, we were introduced to the PCTSP. The major portion of the research we present in this thesis has been carried out on exact and near-optimal solutions of PCTSP, which arises as a subproblem

of SVRP in our column generation approach. In Chapter 3, we model and solve the PCTSP by implementing a branch-and-cut algorithm. We demonstrate that the software is capable of solving instances of the problem efficiently by running various experiments on standard libraries of the VRP instances. The branching mechanism we developed can be used for the SVRP with minor modifications. We model the SVRP problem in this chapter and lay out the solution framework we have designed. We can solve the instances of the SVRP problem by the software, although we should make enhancements to achieve a desirable running time. We briefly explain some possible improvements as future work in the next chapter.

2.2 Column Generation for the SVRP

2.2.1 Preliminaries and Notation

We assume the problem instance is given as a complete (metric) graph $G = (V, E)$, associated costs to the edges c, and the set of drivers D. Recall that the set of vertices, V include the set of customers S and a particular node r called the depot. As before, we let c_{uv} denote the cost of traveling along the edge $e = (u, v)$. We assume that the costs satisfy the triangle inequality; for a set of three customers u, v, and w, we have $c_{uv} + c_{vw} \geqslant c_{uw}$. Each driver i has a set of skills K_i and each customer j has a skill requirement q_j. Driver i can service customer j if $q_j \in K_i$. We also use the following notation in the formulation we provide in Section 2.2.2.

- $\mathcal{T}(i)$ is the set of all tours that can be served by driver i. In other words, the tours in $\mathcal{T}(i)$ can be assigned to the driver i. For every job $j \in \mathcal{T}(i)$, the driver i has the required skill q_j to visit the site.

- $z_{i,T}$ is an indicator binary variable which assumes the value of one if the tour T is to be served by driver i for some $i \in D$, and zero otherwise.

$$z_{i,T} = \begin{cases} 1 & \text{if driver } i \text{ visits the vertices in the tour } T \\ 0 & \text{otherwise} \end{cases}$$

- $C(T)$ is the cost of the tour, *i.e.*, the sum of the costs of its edges.

As a convention, we use **bold** symbols to denote vectors. When dealing with a solution of the linear programming problems, we let z denote the objective function. Also, $\bar{z}$ denotes an upper bound of z, $\underline{z}$ denote a lower bound and z^* specifies the optimal solution. For variables of linear programming x and y, $\hat{x}$ and $\hat{y}$ denote the values of the variables after the LP problem has been solved.

2.2.2 A Formulation using Column Generation

We use a column generation approach to obtain the formulation below.

$$\min \sum_{i,T \in \mathcal{T}(i)} z_{i,T} \cdot C(T) \tag{2.2.1}$$

subject to

$$\sum_{T \in \mathcal{T}(i)} z_{i,T} = 1 \qquad \forall i \in D \tag{2.2.2}$$

$$\sum_{i,T \in \mathcal{T}(i): T \ni j} z_{i,T} \geqslant 1 \qquad \forall j \in S \tag{2.2.3}$$

$$z_{i,T} \in \{0, 1\} \qquad \forall\, i \in D, \forall\, T \in \mathcal{T}(i) \tag{2.2.4}$$

We call the linear programming formulation given in Equations (2.2.1)–(2.2.3) the *master* problem. The objective is to minimize the cost of all tours included in the solution. Constraints 2.2.2 and 2.2.3 are usually referred to as the *assignment* constraints. Constraint 2.2.2 forces the solution to assign any driver to exactly one tour, and Constraint 2.2.3 makes sure that the solution covers every customer with at least one tour. Note that we are minimizing the total cost of the selected tours. As a result, in a metric space, Constraint 2.2.3 is equivalent to

$$\sum_{i,T \in \mathcal{T}(i): T \ni j} z_{i,T} = 1 \qquad \forall j \in S. \tag{2.2.5}$$

The reason is that it would not be beneficial to cover a customer $j \in S$ with more than one tour due to the cost functions.

Remark 1. *The linear programming formulation presented in this section is a well-known formulation with applications to other problems such as scheduling. The formulation is known as the* configuration LP *in the scheduling literature mostly for its classic application in bin packing problems. It has a small integrality gap which makes it suitable for branch-and-bound algorithms as well as for designing approximation algorithms. Indeed, Asadpour et al. used an almost identical formulation to provide the first constant factor approximation algorithm for the Santa Claus problem (see Section 1.4.3).*

Note that the skill set is not explicitly mentioned, but is accounted for implicitly in the sets $\mathcal{T}(i)$. Also, note that the set of all tours has exponential size. Therefore, the integer linear programming problem we wish to solve has an exponential number of variables. A powerful tool to deal with an LP formulation with a large number of variables is the *delayed column generation* (or simply column generation) method. In this method, instead of solving the LP with the large variable set, a smaller formulation is attempted first. The

smaller formulation only considers a subset of the variables while keeping the same set of constraints. Inevitably, the objective value obtained may be sub-optimal. To test the optimality, one can search for a *negative reduced cost column* (variable) in the same fashion the simplex algorithm finds a variable for pivoting. If the search is successful, one can add it to the smaller LP formulation and iterate. If no negative reduced cost column is found, the objective value is provably optimal. The search for a negative reduced cost column typically amounts to solving another problem known as the *subproblem*. Column generation is an effective machinery in settings where the subproblem is relatively easy to solve.

Remark 2. *We may view the delayed column generation method as a branch-and-cut algorithm performed on the dual of a given linear programming formulation. In that light, finding a negative reduced cost column is the separation problem performed to detect violated constraints for the dual.*

In the remainder of this chapter, we argue why column generation is the right line of attack for the SVRP. We will derive the subproblem formulation from the master problem which happens to be a well-known problem in combinatorial optimization – the *prize collecting traveling salesman problem* (PCTSP). PCTSP is the primary focus of Chapter 3 in this thesis. The operations research community has extensive experience in the Traveling Salesman Problem and its variants. As we will demonstrate, we can leverage this experience to develop methods capable of solving the PCTSP efficiently and effectively. These methods are efficient in that they can construct exact solutions to the problem quickly, and are effective since they can also provide sub-optimal solutions which can be used as bounds in pruning the search space. Thus, we can confidently claim that column generation technique is well-equipped for solving the master problem.

Master Problem Formulation

We write the LP relaxed formulation for the master problem as follows.

$$P: \quad \min \sum_{i,T \in \mathcal{T}(i)} z_{i,T} \cdot C(T) \tag{2.2.6}$$

subject to

$$-\sum_{T \in \mathcal{T}(i)} z_{i,T} \geqslant -1 \qquad \forall i \in D \tag{2.2.7}$$

$$\sum_{i,T \in \mathcal{T}(i): T \ni j} z_{i,T} \geqslant 1 \qquad \forall j \in S \tag{2.2.8}$$

$$z_{i,T} \in [0,\, 1] \qquad \forall i \in D,\, \forall T \in \mathcal{T}(i) \tag{2.2.9}$$

We also refer to this formulation as the primal. We will present the dual formulation shortly. As the formulation implies, the goal is to minimize the cost of all the chosen tours in

such a way that **i)** the skill requirements are met, **ii)** each site is covered by at least one tour, and **iii)** each driver visits the nodes of exactly one tour (the equality in Constraint 2.2.7 can be relaxed to less than or equal, meaning that a driver should not serve more than one tour). The Constraint 2.2.9 is relaxed to $z_{i,T} \geqslant 0$ in the linear programming relaxation. We associate dual variables α_i with each constraint $i \in D$, and β_j with each constraint $j \in S$. The dual formulation of the master problem is as follows.

$$D: \qquad \max \sum_{j \in S} \beta_j - \sum_{i \in D} \alpha_i \tag{2.2.10}$$

subject to

$$\sum_{j \in T} \beta_j - \alpha_i \leqslant C(T) \qquad \forall i \in D,\, \forall T \in \mathcal{T}(i) \tag{2.2.11}$$

$$\alpha_i \geqslant 0 \qquad \forall i \in D \tag{2.2.12}$$

$$\beta_j \geqslant 0 \qquad \forall j \in S \tag{2.2.13}$$

Assume that $(\hat{\boldsymbol{\alpha}}, \hat{\boldsymbol{\beta}})$ is the optimal solution to the dual, in which $\hat{\boldsymbol{\alpha}} = (\hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_{|D|})$ and $\hat{\boldsymbol{\beta}} = (\hat{\beta}_1, \hat{\beta}_2, \ldots, \hat{\beta}_{|S|})$ are vectors. As mentioned before, the number of variables (columns) is exponential in the size of set S. To solve the LP relaxation of the above formulation, we start with a number of independent columns equal to the number of rows in the constraint matrix. To obtain the optimal set of columns, we generate columns with negative reduced cost. Specifically, any column with negative reduced cost satisfies the inequality $C(T) < \sum_{j \in T} \hat{\beta}_j - \hat{\alpha}_i$ and can profitably enter the basis. Note how a negative reduced cost column corresponds to a violation of Constraint 2.2.11. Finding a tour T that satisfies this condition is the objective of the subproblem. Alternatively, we seek a tour T for driver $i \in D$ such that $\sum_{j \in T} \hat{\beta}_j - C(T) > \hat{\alpha}_i$. We accomplish this by maximizing $\sum_{j \in T} \hat{\beta}_j - C(T)$ for each driver $i \in D$. Note that $\hat{\alpha}_i$'s are constant values and we do not need to include them in the objective function of the subproblem. By maximizing $\sum_{j \in T} \hat{\beta}_j - C(T)$, we are indeed finding the constraint in the dual that is violated with the highest margin. In fact, one can settle for finding any tour for which $C(T) < \sum_{j \in T} \hat{\beta}_j - \hat{\alpha}_i$. Such a tour still corresponds to a negative reduced cost column which may potentially improve the objective value should it be added to the basis. Still, there are other benefits to finding the largest negative reduced cost in the column generation scheme which justify the effort. Most notably, a column with the highest negative reduced cost can provide us with a *dual bound*.

The Dual Bound

The concept of a dual bound suggests that by optimizing the subproblem, one can modify the dual variables to obtain a dual feasible solution, which in turn can provide us with a lower bound for the master problem. Note that since we solve the master problem on a subset

of variables, the corresponding $(\hat{\boldsymbol{\alpha}}, \hat{\boldsymbol{\beta}})$ is not necessarily feasible in the dual. As we stated earlier, a negative reduced cost column in the primal is indeed a violated constraint in the dual. Let z^* denote the optimal primal solution (to the master problem) and $(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*)$ denote the optimal for the dual (subproblem). Also assume $f(z)$ and $g(\boldsymbol{\alpha}, \boldsymbol{\beta})$ are the objective functions of the primal and the dual respectively. We know that $g(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*) \leqslant f(z^*)$ from weak duality. Since $g(\hat{\boldsymbol{\alpha}}, \hat{\boldsymbol{\beta}}) \leqslant g(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*)$ for any feasible solution $(\hat{\boldsymbol{\alpha}}, \hat{\boldsymbol{\beta}})$, $g(\hat{\boldsymbol{\alpha}}, \hat{\boldsymbol{\beta}})$ is a lower bound for the primal. Lemma 1 shows how to obtain a dual feasible solution for our vehicle routing problem.

Lemma 1. *Let $(\hat{\boldsymbol{\alpha}}, \hat{\boldsymbol{\beta}})$ be a dual solution for the subproblem solved on a subset of constraints. Then, $(\hat{\boldsymbol{\zeta}}, \hat{\boldsymbol{\beta}})$ is a feasible solution for the dual problem, in which $\hat{\boldsymbol{\zeta}} = (\hat{\zeta}_1, \hat{\zeta}_2, \ldots, \hat{\zeta}_{|D|})$. For every driver $i \in D$ we have:*

$$\hat{\zeta}_i = \begin{cases} \max\{\sum_{j \in T} \hat{\beta}_j - C(T)\} & \textit{if driver } i \textit{ has a negative reduced cost column } T \\ \hat{\alpha}_i & \textit{otherwise} \end{cases}$$

Proof. If driver i does not have a negative reduced cost column, then $\sum_{j \in T} \hat{\beta}_j - C(T) \leqslant \hat{\alpha}_i$. Therefore, $\hat{\alpha}_i$ is dual feasible for i. Now, consider a driver i who has a negative reduced cost column, meaning that a tour $T \in \mathcal{T}(i)$ exists that for which Constraint 2.2.11 is violated. Assume that $T^* \in \mathcal{T}(i)$ is the tour for which $\sum_{j \in T*} \hat{\beta}_j - C(T^*) = \hat{\zeta}_i$ is maximized. Also assume T' is any other negative reduced cost column for i. Since the driver i has a negative reduced cost column, then $\hat{\zeta}_i \geqslant \sum_{j \in T'} \hat{\beta}_j - C(T') > \hat{\alpha}_i \geqslant 0$. By rearranging the last inequality, we also get $\sum_{j \in T'} \hat{\beta}_j - \hat{\zeta}_i \leqslant C(T')$. Therefore, $\hat{\boldsymbol{\zeta}}$ satisfies both the lower bound and upper bound conditions for the dual, and $(\hat{\boldsymbol{\zeta}}, \hat{\boldsymbol{\beta}})$ is dual feasible. □

For every driver i with a negative reduced cost column (a violated constraint), we increase $\hat{\alpha}_i$ to $\hat{\zeta}_i$ to get a dual feasible solution while we do not change $\hat{\alpha}_i$ for other drivers. Since α appear as a negative term in the objective function of the dual problem, the increase in its value decreases the objective value of the solution. The amount of decrease is $\sum_{i \in D} \hat{\zeta}_i - \hat{\alpha}_i$.

Subproblem Formulation

The subproblem we solve to extract a negative reduced cost column is the famous Prize Collecting Traveling Salesman Problem or PCTSP for short. PCTSP is a well-known NP-hard problem that has received a lot of attention in the literature. We give below a common ILP formulation of the subproblem. [89].

$$\max \sum_{j \in S} \beta_j y_j - \sum_{e \in E} c_e x_e \tag{2.2.14}$$

subject to

$$\sum_{e \in \delta(i)} x_e = 2y_i \qquad \forall i \in S \tag{2.2.15}$$

$$\sum_{e \in E(V)} x_e \leqslant \sum_{i \in V \setminus \{k\}} y_i \qquad \forall k \in V, V \subseteq S' \tag{2.2.16}$$

$$y_r = 1 \tag{2.2.17}$$

$$x_e \in \{0,1\}, y_j \in \{0,1\} \qquad \forall e \in E, \forall j \in S \tag{2.2.18}$$

The integer variable y_j is set to 1 (0) if node $j \in S$ is included (not included) in the tour. Similarly, the integer variable x_e is set to 1 (0) if edge $e \in E$ is included (not included) in the tour. Note that $\sum_{j \in S} \beta_j y_j = \sum_{j \in T} \beta_j$ and $\sum_{e \in E} c_e x_e = \sum_{e \in T} c_e = C(T)$. Constraint 2.2.15 ensures that if node j is included in the tour, then two edges of the tour must be incident on it. Constraint 2.2.16 is the generalized sub tour elimination constraint (GSEC). Constraints of this form are used to prevent any sub-tour that does not include the root node r. Chapter 3 gives an in-depth discussion on a branch-and-cut method for solving the PCTSP.

2.3 The Branch-and-Price Framework

Here, we briefly skim the general procedure of the branch-and-price algorithm. The algorithms described in the two chapters of this part of the thesis are branch-and-bound algorithms in nature. They work around searching the tree formed by all possible values of the decision variables and use bounds to prune as much of the search space as possible. The difference between the two algorithms is in the way they obtain the lower and upper bounds. The branch-and-price algorithm is described in Figure 2.1. The algorithm has many parts in common with the branch-and-cut algorithm that we use for the PCTSP in Chapter 3. Therefore, we defer a more detailed discussion of those parts to the next chapter.

The branch-and-price algorithm takes the following steps upon execution:

Step 1. Find an integral solution to the master problem (denoted by MP in the flowchart) through some heuristic algorithm such as the local search. The objective value of any integral feasible solution can be used as an upper bound. We let $\bar{z}$ denote this upper bound.

Step 2. Generate the first LP formulation for the master problem. This formulation is given in Section 2.2.2. The algorithm maintains a list L of unsolved problems. The formulation then is added to L as the first node of the branching tree.

Step 3. Retrieve the front of the list L and use CPLEX to solve it. We let z^* denote the objective value of the optimal (fractional) solution to the LP. If the value is less than

the lower bound, then fathom the current node. By fathoming a node, we mean we decide not to explore its sub-tree further since we can fathom the best objective value the sub-tree can provide by examining the bounds and z^*.

Step 4. Check if the fractional solution can be rounded to an integral one that improves the upper bound. If so, update the lower bound and reassess the problem for fathoming. Otherwise, continue to Step 5.

Step 5. Remove the columns that have not been in the basis for a preset number of past rounds. We set a threshold for the number of rounds a column is not used in the basis. These columns correspond to variables that are not effective in decreasing the objective value. Therefore, we can remove them from the variables of the formulation for now, and if they ever become effective, the column generation phase detects them and adds them back to the model. In removing the variables, we take care not to remove the ones we have fixed through branching.

Step 6. Generate columns; The column generation leads us to the subproblem, the PCTSP, which is the subject of Chapter 3. After solving the subproblem, we obtain the values $\hat{\boldsymbol{\zeta}} = (\hat{\zeta}_1, \hat{\zeta}_2, \ldots, \hat{\zeta}_{|D|})$. Using this vector, we can check if $z^* - \sum \hat{\boldsymbol{\zeta}}$ gives a better lower bound for the problem. If yes, we update the lower bound, denoted by $\underline{z}$. This bound can be used as a criterion for terminating the column generation process and moving to the branching phase. One obvious criterion is if no more columns with negative reduced cost columns exist. Through experimentation, we have learned that often, the solution to the subproblem does provide the master problem with negative reduced cost columns. But the lower bound and upper bound on z^* are so close that the improvements we make are insignificant. Therefore, it is beneficial to stop generating columns and go the next step in these situations.

Step 7. If there are any variables with fractional values in the solution, choose a variable for branching and create two new nodes P_0 and P_1. We set the variable to 0 in P_0 and to 1 in P_1. We add the two nodes P_0 and P_1 to the list L of the problems and go to Step 3 again. If no fractional values are left, then we fathom the node and go back to Step 3. The algorithm terminates and reports back the incumbent (the integral solution corresponding to the best upper bound) if the list is empty.

The Next Step

We postpone further discussions about the solution framework we have devised to Chapter 3. In the next chapter, we go into the details of the solution method that we used for the subproblem and provide experimental evidence to argue its efficiency. We conclude the chapter with a summary of what we have built in these two chapters, as well as provide some explanation on how we plan to build upon our current solution framework to tackle real-world instances of the Skill Vehicle Routing Problem.

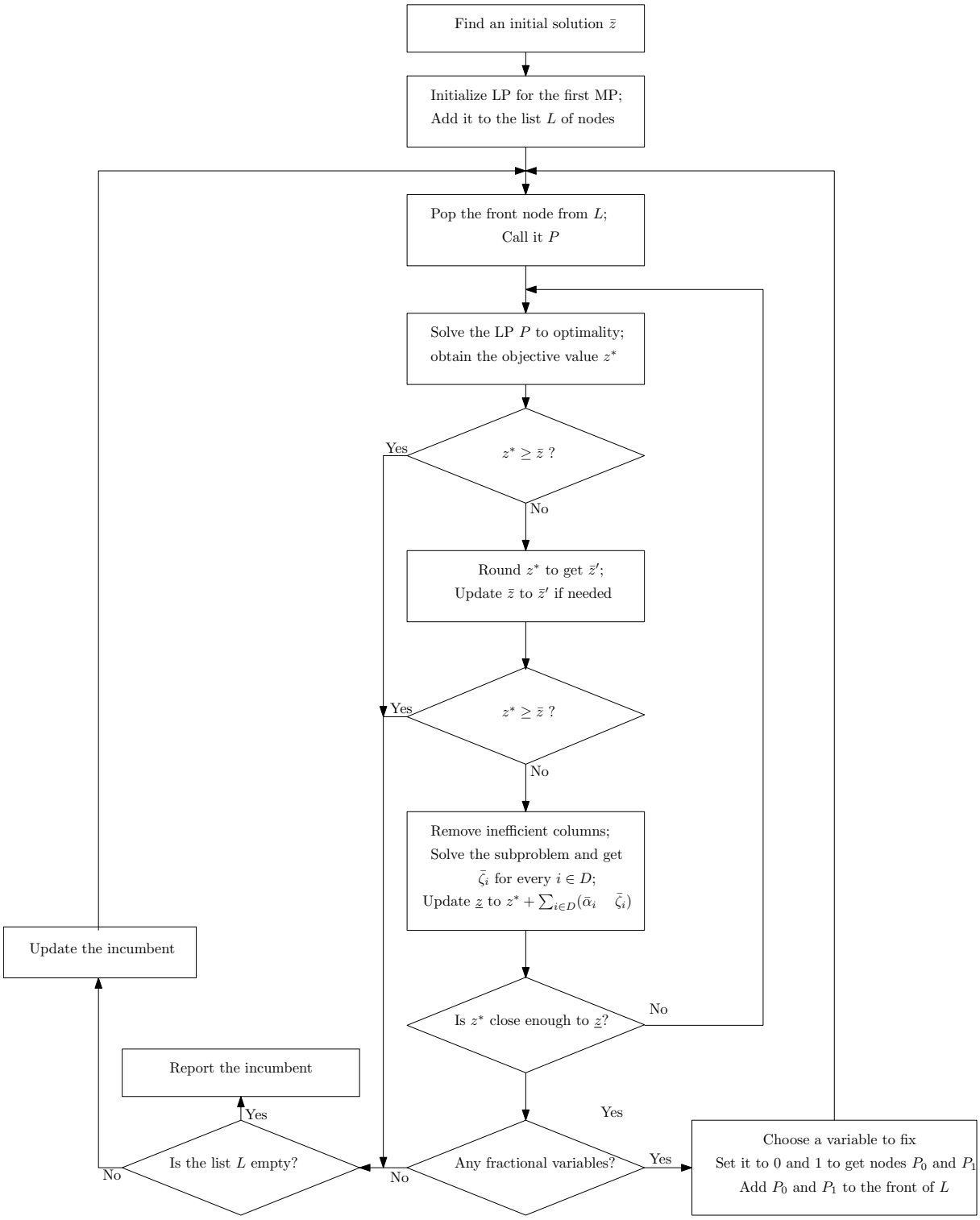


Figure 2.1: The Branch-and-Price Algorithm

Chapter 3

A Branch-and-Cut Algorithm for Prize Collecting Traveling Salesman Problem

3.1 Introduction and Problem Definition

The Prize Collecting Traveling Salesman Problem (PCTSP) is a variant of the Traveling Salesman Problem (TSP). Given a set of locations and a cost function for travel times between them, The Traveling Salesman Problem asks for the cheapest tour that visits every location exactly once. Being one of the oldest and most well-known problems in combinatorial optimization, the TSP has inspired studies in computer science, mathematics, chemistry, physics, and psychology among other areas, and has applications in logistics, manufacturing, genetics, telecommunications, neuroscience, and more [4]. Throughout the years, researchers have studied variations of the TSP. The Prize Collecting TSP is one which is important in itself [108]. It also arises as a subproblem in column generation formulations of various vehicle routing problems. As discussed in Chapter 2, the solution to the PCTSP gives a column with a reduced cost.

The PCTSP comprises a complete graph $G = (S, E)$, where S is a set of sites and E is the set of edges. There is a special node called the *depot*, denoted by r. Also given is a set a cost function $c : E \to \mathbb{R}_{>0}$ which represents the travel distance between the nodes. In addition to costs, each site has a *prize* associated with it. We let the function $\beta : S \to \mathbb{R}_{>0}$ denote the prizes. The objective is to derive a tour which includes the depot and maximizes the sum of the prizes associated with the nodes in the tour, less the cost of the tour. It can be considered as a generalization of the TSP since the TSP is the PCTSP with a high enough prize associated with each node.

3.1.1 Related Work

PCTSP is an NP-hard problem and has received a lot of attention in the literature. Several variants of the PCTSP have been studied in the literature. The version of the PCTSP we study in this research was first introduced by Balas [8] in a more general setting to model the scheduling of the daily activities of a steel rolling mill. In the version that Balas introduced, a tour can profit (to the extent of a prize associated with the node) from each node it visits, while it is penalized (to the extent of a penalty associated with the node) for every node it does not visit. The profit/penalty for each node corresponds to the prize associated with the node. The objective is to obtain a tour such that the total prize collected exceeds a prescribed amount while minimizing the sum of the travel cost and penalties. The travel cost for a tour is the sum of the distances between consecutive nodes in the tour. Balas [8] derives the cuts for the PCTSP corresponding to the subtour elimination constraints for the TSP. The cuts we use in this thesis include a class of constraints knows as the Generalized Subtour Elimination Constraints (GSECs). Goemans first applied the GSECs in the context of the Steiner tree problem [51]. See also Wolsey [108] for a precise treatment of our version of the PCTSP. In a continuation of his work on PCTSP, Balas [9] derives, among other cuts, the cuts corresponding to the primitive comb inequalities for the TSP. The separation heuristic we use for these cuts is a heuristic given by Padberg and Hong [93] for detecting blossoms for the TSP. We show that these heuristics can also be used as separation procedures for the primitive comb inequalities for the PCTSP. We use these heuristics to obtain LP solutions with improved integrality gap.

The first known solution procedure to solve the PCTSP exactly was a branch-and-bound procedure [44]. Fischetti et al. also study a branch-and-cut algorithm for the symmetric case of Generalized Traveling Salesman problem, in which the nodes are split into clusters, and any feasible solution to the problem should cover at least one node from each cluster [43]. The symmetric property refers to the fact that the cost of going from a node u to another node v is the same is the cost of v to u. Furthermore, Fischetti et al. use a similar branch-and-cut algorithm for the orienteering problem [42]. In an orienteering problem, a prize is associated with each vertex. An optimal tour must collect the maximum prize while keeping the travel cost below a given threshold. In more recent work, Chaves and Lorena propose a hybrid metaheuristic for the problem and compare its performance with CPLEX [23]. Bérubé, Gendreau, Potvin propose a branch-and-cut algorithm to solve a variant of the PCTSP [12]. In this variant, the objective is to obtain a tour with minimum travel cost, subject to the constraint that the total prize collected exceeds a predetermined amount. They incorporate many valid inequalities for their variant of the PCTSP and evaluate the performance of a branch-and-cut algorithm by introducing these inequalities.

3.1.2 Our Contribution

Our version of the PCTSP arises directly as a subproblem in a variant of the vehicle routing problem, called the Skill Vehicle Routing Problem [19]. In this chapter, we focus on the LP solution procedure for this version of the PCTSP. We develop a branch-and-cut algorithm to seek the exact solution of the problem. A local search heuristic is used to both find an initial feasible solution as well as improve the fractional solutions obtained from the LP relaxation of the problem. Then, we introduce a set of cuts to the setting, valid for any feasible integral solution, to chop off some fractional solution and improve the objective value. Furthermore, we use fast separation heuristics for both the GSECs and the primitive comb inequalities. For the GSECs, we adapt a fast heuristic, and compare its performance with an exact procedure, both in the quality of its solution, as well as in its running time. For the primitive comb inequalities, we show that a well-known separation heuristic for the TSP [93] also works as a separation procedure for the PCTSP. We also compare the GSEC cuts with the primitive comb inequalities in their quality of solutions. Finally, we use some branching heuristics to fix some of the integral variables when there are no more effective cuts that can significantly improve the solution.

3.2 Linear Programming Formulation

3.2.1 Preliminaries and Notation

We let $G = (S, E)$ denote the complete graph representing the instance of the problem, with node $r \in S$ indicating the depot which every tour must visit. Cost c_e associated with each edge $e \in E$ is the Euclidean distance between the two endpoints of edge e. In the PCTSP, the salesman may choose to visit city $j \in S$. If he does so, then he obtains a prize β_j but incurs a travel cost c_e if he traverses edge $e = (i, j)$. The salesman must start and end his tour at node r, and maximize the total prize he collects, less the cost of the tour.

Throughout this chapter, we use x to refer to the decision variables of edges, and y and z to refer to the decision variables of vertices. For a set of vertices V, $y(V)$ denotes the sum of y's for all the nodes of V. For simplicity, we let y_v denote $y(\{v\})$. For a set of vertices S and a subset V of S, $S \setminus V$ represents $S - V = \{v : v \in S \wedge v \notin V\}$. We say that $(S, S \setminus V)$ represents a *vertex cut*. Also, for any two sets of vertices A and B, $E(A, B)$ denotes the set of edges in the cut (A, B), that is, those edges which have exactly one endpoint in A and another in B.

3.2.2 Problem Formulation

We provide below the ILP formulation for the PCTSP [108]. In the formulation below, we let decision variables y_j be 1 if the salesman visits city j (and 0 otherwise), and x_e be 1 if he

traverses edge e (and 0 otherwise). We also let set $S' = S\backslash\{r\}$ and $E' = E\backslash\{\delta(r)\}$, where $\delta(r)$ is the set of edges incident on the depot node r.

$$\max \sum_{j\in S} \beta_j y_j - \sum_{e\in E} c_e x_e \tag{3.2.1}$$

subject to

$$\sum_{e\in\delta(i)} x_e = 2y_i \qquad \forall i \in S \tag{3.2.2}$$

$$\sum_{e\in E(V)} x_e \leqslant \sum_{i\in V\backslash\{k\}} y_i \qquad \forall k \in V, V \subseteq S' \tag{3.2.3}$$

$$y_r = 1 \tag{3.2.4}$$

$$x_e \in \{0,\, 1\},\ y_j \in \{0,\, 1\} \qquad \forall e \in E, \forall j \in S \tag{3.2.5}$$

The integer variable y_l is set to 1 (0) if node $l \in S$ is included (not included) in the tour. Similarly, the integer variable x_e is set to 1 (0) if edge $e \in E$ is included (not included) in the tour. Note that for a tour T selected in the ILP solution we have that $\sum_{j\in S} \beta_j y_j = \sum_{j\in T} \beta_j$ and $\sum_{e\in E} c_e x_e = \sum_{e\in T} c_e = C(T)$. Constraint 3.2.2, known as the *degree constraint*, ensures that if node l is included in the tour, then two edges of the tour must be incident on it. Constraint 3.2.3 is the generalized subtour elimination constraint (GSEC). Constraints of this form are used to prevent any sub tour that does not include root node r. These are generalizations of the subtour elimination constraints for the standard traveling salesman problem. In the traveling salesman polytope, we also need to avoid subtours as we seek a single cycle that covers all the vertices. Since all the vertices are selected in the cycle, inequalities of the following form are introduced:

$$\sum_{e\in E(V)} x_e \leqslant |\, V \,| - 1 \qquad \forall V \subseteq S' \tag{3.2.6}$$

Constraint 3.2.6 is known as the Subtour Elimination Constraint (SEC). In the prize-collecting polytope, we have the flexibility to cover only a subset of the vertices through y_i variables. Therefore, Constraint 3.2.6, although still a valid inequality, may not be tight in the setting of the PCTSP. Balas generalized the notion of SEC to obtain the GSECs of the form of Constraint 3.2.3 [8]. Clearly, there are an exponential number of constraints included in the GSECs and we cannot include them all to solve the subproblem. To get around this problem, we include these constraints as they are needed (whenever there is a subtour that does not include node r). It is easy to detect such subtours when the decision variables take 0/1 values. However, because we solve the LP relaxation of the subproblem, fractional values may be assigned to the decision variables y_j and x_e. We outline below the

method that can be used to detect for such sub tours (which do not include node r) when fractional values are assigned to the decision variables.

Let the LP solution to the sub problem be given by $(x*, y*)$. Then the generalized sub tour elimination constraint is violated for a subset $W \subseteq S'$ of nodes if the inequality $\sum_{e \in E'(W)} x_e^* > \sum_{l \in W \setminus \{k\}} y_l^*$ is satisfied. Such a subset W can be extracted by solving the integer program given by the Objective Function (3.4.1) and Constraints 3.4.2–3.4.4. We call the problem below the separation problem for GSECs. The formulations for the separation problem for GSECs for the PCTSP are given in Wolsey 1998 [108].

3.3 The Branch-and-Cut Algorithm

In this section, we describe different steps of the branch-and-cut algorithm. In the subsequent sections, we will discuss each step in detail separately. Figure 3.1 depicts the flowchart of the algorithm. In the flowchart, z^* represents the solution of the LP relaxation of the problem, and $\underline{z}$ is the integral solution, which we obtain every time the *incumbent* is updated. An incumbent is the best feasible integral solution found so far at any stage of a run of the algorithm. The incumbent may change either when the local search finds a better solution or when the solution to the LP relaxation at a node is integral.

Step 1. Use a local search heuristic algorithm to create a feasible solution. A feasible solution is a tour that contains the depot and visits a subset of the nodes exactly once, entering with one edge and leaving with another. Note that since the problem is formulated as a maximization, any feasible solution serves as a lower bound. The solution produced by the local search is the first incumbent for the branch-and-cut algorithm. $\underline{z}$ denotes the objective value of the incumbent.

Step 2. Generate the first LP relaxation of the problem. In this step, we set up a linear programming problem with the objective function (2.2.14), the set of degree constraints (Constraint 3.2.2), and the restriction that the solution should contain the depot (Constraint 3.2.4). Note that Constraint 3.2.5 is replaced by

$$x_e \in [0, 1],\ y_j \in [0, 1] \qquad \forall e \in E, \forall j \in S \tag{3.3.1}$$

in the LP relaxation. We use a list L to keep track of the problems to be solved for the branch-and-cut algorithm. The LP relaxation described here is placed in L as the first problem.

Step 3. Retrieve the front of the list L and use CPLEX Linear Programming solver to solve it. We let z^* denote the optimal (fractional) solution to the LP. At any point in the execution of the algorithm, we might get a z^* value that is smaller than the objective value of the incumbent denoted by $\underline{z}$. The reason is that as we progress, we introduce more constraints to the model and fix a subset of variables, which can significantly restrict the

solution space. In such cases, we will not benefit from continuing this branch of the problem since every node can be considered a relaxation of all the subsequent nodes in its subtree. Therefore, if $z^* \leqslant \underline{z}$, we *fathom* the problem and repeat STEP 3 if any other problem exists in the list L. If the list L is empty, we have completed the branching and can report the incumbent as the optimal solution.

STEP 4. If $z^* > \underline{z}$, then check if we can improve the incumbent by performing the local search again. The only difference with this application of the local search is that we run it on the set of vertices whose y value is larger than a threshold. Through experimentation, we have chosen a threshold value of 0.3. If the local search improves the incumbent, we update the value of $\underline{z}$, and check if $z^* \leqslant \underline{z}$ once more. If $z^* \leqslant \underline{z}$, we should fathom this problem and repeat STEP 3.. Otherwise, go to STEP 5.

STEP 5. Remove the constraints that have not been tight in the past five rounds at this node. We look at the slack variables after each solution of the LP relaxation at a node. If the slacks are strictly positive, it means that the corresponding constraints have not been tight, and therefore, do not impact the solution. We keep track of a *redundancy* counter for each constraint. If the counter reaches a preset threshold of five, we remove the constraints from the model. Constraint removal may sound counter-intuitive at first since we usually restrict the solution space further as we move down the branch-and-cut tree while removing cuts may relax the space. The rationale behind constraint removal is that the running time for most of the linear programming solvers, including CPLEX that we have used in this research, is sensitive to the number of constraints. The running time can drastically increase with the number of constraints. Therefore, adding many constraints to a model is computationally expensive. If these constraints happen to be redundant in the sense that they do not affect the solution by much, one can usually benefit from removing them from the model. Our solution will stay valid, and if any of the removed cuts becomes violated in a later iteration, the constraint generation methods can detect it and add it back to the model. In removing the cuts, we take care not to remove constraints corresponding to fixing variables.

STEP 6. Generate cuts; detect up to 50 violated GSECs and 50 violated Primitive Comb inequalities and add them to the model. The threshold of 50 is set to ensure we do not add too many cuts to the model at once, which can potentially increase the number of redundant cuts. To detect violated cuts quickly, we have adapted to heuristic algorithms used for similar cuts of the TSP. We will explain these heuristic algorithms in Sections 3.4 and 3.5. If the heuristics find any violated cuts, we then solve the LP relaxation of this new model and compare z^* to $\underline{z}$ as in STEP 3. Otherwise, we go to the branching step.

STEP 7. If there are any variables with fractional values in the solution, choose a variable for branching and create two new nodes P_0 and P_1. In P_0, we set the variable to 0 and in P_1 to 1. The choice of the variable to *fix* hugely impacts the running time of the algorithm. Ideally, we seek variables for branching that result in balanced branch-and-cut trees. A balanced branch-and-cut tree has a logarithmic height, meaning that the algorithm needs to

traverse a logarithmic number of nodes before reaching a leaf, which is a fathomed node. A fathomed node can be of the following two types: **i)** either a node with an integral solution, which can potentially update the incumbent and help with the pruning of the tree, or **ii)** have a z^* higher than the objective value of the incumbent, which means we do not need to pursue them any further. In any case, it is desirable for us to reach the leaves faster. An efficient heuristic for branching of the prize-collecting TSP is due to Gendreau et al. [48]. We explain this heuristic in Section 3.6. We add the two nodes P_0 and P_1 to the list L of the problems and go to STEP 3 again. If no fractional values are left, then we have arrived at an integral solution. We can now fathom the node and go back to STEP 3 to solve the next problem in the list L. The algorithm terminates and reports back the incumbent if the list is empty.

3.4 Generalized Subtour Elimination Constraints

A formulation for the separation problem for GSECs is given below.

$$\zeta_k = max \sum_{e \in E'} x_e^* w_e - \sum_{j \in W \setminus \{k\}} y_j^* z_j \tag{3.4.1}$$

subject to

$$w_e \leqslant z_i, w_e \leqslant z_j \qquad \forall e = (i,j) \in E' \tag{3.4.2}$$

$$w_e \geqslant z_i + z_j - 1 \qquad \forall e = (i,j) \in E' \tag{3.4.3}$$

$$w_e \in \{0,1\}, z_j \in \{0,1\}, z_k = 1 \qquad \forall e \in E', \forall j \in W. \tag{3.4.4}$$

Constraint 3.4.3 above can be dropped because it is implied by Constraint 3.4.2 if $x_e^* \geqslant 0$ for $e \in E'$ [108]. It turns out that the constraint matrix for the above separation problem (after Constraint 3.4.3 is dropped) is totally unimodular. Thus, in polynomial time, we solve the LP relaxation of the separation problem to obtain constraint(s) to introduce into the sub problem.

3.4.1 The Modified Separation Problem for GSECs

We write the modified separation problem below.

$$\zeta_k = max \sum_{e \in E'} x_e^* w_e - \sum_{j \in W \setminus \{k\}} y_j^* z_j \tag{3.4.5}$$

subject to

$$w_e \leqslant z_i, w_e \leqslant z_j \qquad \forall e = (i,j) \in E' \tag{3.4.6}$$

$$w_e \in \{0,1\}, z_j \in \{0,1\}, z_k = 1 \qquad \forall e \in E', \forall j \in W. \tag{3.4.7}$$

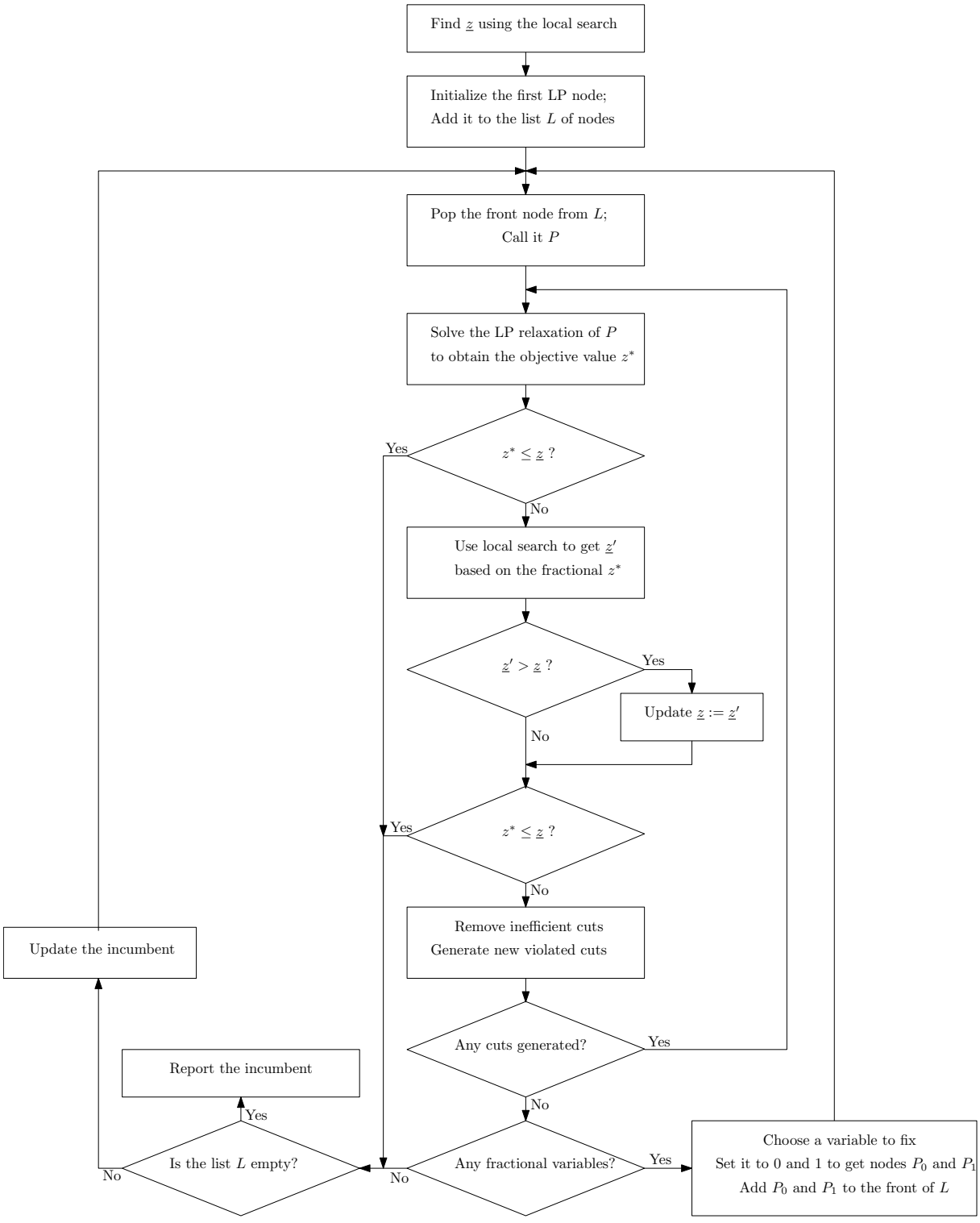


Figure 3.1: The Branch-and-Cut Algorithm

The optimal solution to the LP relaxation of the modified separation problem is integral. This optimal solution corresponds to the subset $W \subseteq S'$ and node $k \in W$ such that the inequality $\sum_{e \in E'(W)} x_e^* \leqslant \sum_{l \in W \setminus \{k\}} y_l^*$, corresponding to Constraint 3.2.3 of the sub problem is violated. This inequality is introduced into the sub problem and the LP relaxation of the PCTSP is again solved. This is repeated until there is no subset $W \subseteq S'$ and node $k \in W$ such that the inequality $\sum_{e \in E'(W)} x_e^* \leqslant \sum_{l \in W \setminus \{k\}} y_l^*$ is violated.

In practice, solving an LP for each $k \in W$ can be very time consuming. Therefore, a *shrinking* heuristic is used to speed up the separation algorithm. The shrinking heuristic is described below.

3.4.2 A Shrinking Heuristic for the Separation of GSECs

The shrinking heuristic we describe below helps find many violated GSECs quickly. We generalize the heuristic developed by Crowder and Padberg [28] and Land [77] for the TSP. Recall that $S' = S \setminus \{r\}$ and $E' = E \setminus \{\delta(r)\}$. In this approach, the GSEC inequalities are transformed into the equivalent cut-set inequalities. In the following, $E(V_1, V_2)$ is used to denote the set of edges which have one endpoint in V_1 and the other in V_2.

$$\sum_{e \in E(V, S' \setminus V)} x_e \geqslant 2y_k \qquad \forall k \in V, V \subseteq S' \tag{3.4.8}$$

Note that the cut-set inequalities can be derived from Constraints 3.2.2 and 3.2.3. These inequalities allow us to transform the problem to a network flow problem in which we seek a minimum cut in a rather special sense. We are looking for violated cut-set constraints. Equivalently, we wish to find a subset $V \subseteq S'$ of vertices for which, $\sum_{e=(i,j),\, i \in V,\, j \in S' \setminus V} x_e < 2y_k$ for some $k \in V$. Alternatively, we look for a cut $(V, S' \setminus V)$ that minimizes

$$\sum_{e \in E(V, S' \setminus V)} x_e - 2 \max_{k \in V} \{y_k\}. \tag{3.4.9}$$

If there exists a k for which Inequality 3.4.8 is violated, then the value of a cut that minimizes 3.4.9 is negative. (If there is no k for which Inequality Inequality (3.4.8) is violated, then the value of any cut that minimizes 3.4.9 is positive.) To find such a cut, we repeatedly reduce the size of the graph by shrinking subsets of vertices into a single vertex, provided that certain conditions are met. Indeed, the shrinking process guarantees that the vertices shrunk into a single vertex will end up on the same shore of the minimum cut described earlier. Consider to subsets of vertices $V_1, V_2 \subseteq S'$. Suppose that, by induction, we know that all the vertices in V_1 will end up on the same shore of the cut, and also the same holds for all the vertices of V_2. If Inequality 3.4.12 holds, there *exists* a minimum cut in which

V_1 and V_2 belong to the same component and we can merge (shrink) them into one single component, since we are not increasing the capacity of the cut by adding V_2 to V_1.

Consider two subsets, $V_1, V_2 \subseteq S'$. If Inequality 3.4.10 below holds, then subsets V_1 and V_2 can be merged.

$$\sum_{e \in E(V_1,\, S' \setminus V_1)} x_e - 2 \max_{k \in V_1} \{y_k\} \geqslant \sum_{e \in E(V_1 \cup V_2,\, S' \setminus (V_1 \cup V_2))} x_e - 2 \max_{k \in V_1 \cup V_2} \{y_k\}. \tag{3.4.10}$$

We will explain this inequality using Figure 3.2.

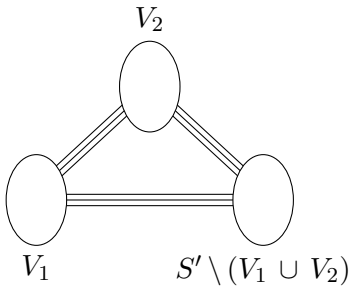


Figure 3.2: The condition for shrinking the set V_2 into the set V_1.

From the perspective of V_1, the value of cut separating V_1 from $S' \setminus V_1$ is the sum of X_e values of all the edges in the cut minus two times the maximum y_k value of any node $k \in V_1$ (the left-hand-side of Inequality (3.4.10)). If it were to merge with V_2, then the sum of x_e values of edges in $E(V_1,\, V_2)$ (edges going from V_1 to V_2) would be replaced by the sum of x_e values of edges in $E(V_2,\, S' \setminus (V_1 \cup V_2))$. At the same time, the maximum of the y_k values is taken over all the nodes $k \in V_1 \cup V_2$. Therefore, if the left-hand-side of Inequality (3.4.10) is no less than the right-hand-side of the same inequality, we will not increase the objective function value of 3.4.9 by putting V_1 and V_2 on the same shore of the cut. Note that the same must hold from the perspective of V_2 as well, else we may increase the objective function value for the component V_2 by merging it with V_1. We rewrite Inequality (3.4.10) above as Equation (3.4.11) below.

$$\sum_{e \in E(V_1,\, V_2)} x_e + \sum_{e \in E(V_1,\, S' \setminus (V_1 \cup V_2))} x_e - 2 \max_{k \in V_1} \{y_k\} \geqslant \sum_{e \in E(V_1,\, S' \setminus (V_1 \cup V_2))} x_e + \sum_{e \in E(V_2,\, S' \setminus (V_1 \cup V_2))} x_e - 2 \max_{k \in V_1 \cup V_2} \{y_k\}. \tag{3.4.11}$$

Subtracting the common term $\sum_{e \in E(V_1, S' \setminus (V_1 \cup V_2))} x_e$ from both sides of Inequality (3.4.11) above, we get Inequality (3.4.12) below.

$$\sum_{e \in E(V_1, V_2)} x_e - 2 \max_{k \in V_1} \{y_k\} \geqslant \sum_{e \in E(V_2, S' \setminus (V_1 \cup V_2))} x_e - 2 \max_{k \in V_1 \cup V_2} \{y_k\}. \tag{3.4.12}$$

Also, we know from Constraint 3.2.2 that

$$\sum_{e \in E(V_2, S' \setminus (V_1 \cup V_2))} x_e = 2 \sum_{i \in V_2} y_i - \sum_{e \in E(V_1, V_2)} x_e - 2 \sum_{x_e \in E(V_2)} x_e. \tag{3.4.13}$$

Therefore, we can shrink V_1 and V_2 if

$$\sum_{e \in E(V_1, V_2)} x_e \geqslant \sum_{i \in V_2} y_i + \max_{k \in V_1} \{y_k\} - \max_{k \in V_1 \cup V_2} \{y_k\} - \sum_{e \in E(V_2)} x_e. \tag{3.4.14}$$

The Inequality (3.4.14) is the basis of our shrinking heuristic. We start with the original graph $G = (S', E')$, and consider pairs of vertices for shrinking. Note that, as we proceed with the shrinking algorithm, a vertex v_i may represent a set of vertices in the original graph G. Thus, we associate a value m_i to every vertex of the graph v_i which denotes the maximum value of y_k's in the corresponding component of v_i in the original graph. For the vertices of the original graph, we set $m_i = y_i$. Now, for every edge $e = (i, j)$ we shrink the two endpoints v_i and v_j if this process would not increase the cuts values from the perspective of each of them, meaning that we shrink v_i and v_j into a single super-node if

$$x_e \geqslant y_j - \max\{0, m_j - m_i\}, \tag{3.4.15}$$

and

$$x_e \geqslant y_i - \max\{0, m_i - m_j\}. \tag{3.4.16}$$

Note that the Inequality (3.4.14) can be written as

$$\sum_{e \in E(V_1, V_2)} x_e \geqslant \sum_{i \in V_2} y_i - \Delta - \sum_{e \in E(V_2)} x_e, \tag{3.4.17}$$

where $\Delta = \max_{k \in V_1 \cup V_2} \{y_k\} - \max_{k \in V_1} \{y_k\}$. Regarding different values of Δ, two cases might occur: **i)** $\max_{k \in V_1} \{y_k\} = \max_{k \in V_1 \cup V_2} \{y_k\}$, in which case $\Delta = 0$, or **ii)** $\max_{k \in V_1} \{y_k\} < \max_{k \in V_1 \cup V_2} \{y_k\}$, meaning that $\Delta = \max_{k \in V_2 \setminus V_1} \{y_k\} - \max_{k \in V_1} \{y_k\}$. These two cases have been considered in the second terms on the right-hand side of Inequalities (3.4.15) and

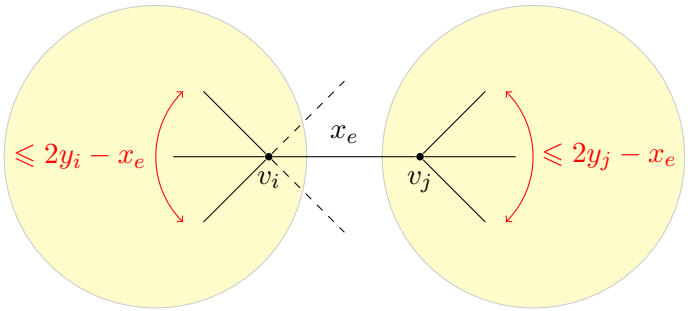


Figure 3.3: Merging to super-nodes y_i and y_j into a single super-node

(3.4.16). Furthermore, whenever we merge two vertices v_i and v_j connected via edge e into a new vertex $v_{i'}$, we are indeed joining two components in the original graph represented by v_i and v_j, connected by a cut denoted by the edge e in the shrunk graph. Therefore, we set the new values $y_{i'} = y_i + y_j - x_e$ and $m_{i'} = \max\{m_i, m_j\}$. The value $m_{i'}$ denotes the maximum value of y_i^*'s originally obtained from the LP solution. Note that $y_{i'}$ shows the capacity of the new node $v_{i'}$ in some sense, i.e., half of the sum of all edge weights connected to $v_{i'}$. This is due to the degree constraints. The sum of edges going out of the component of y_i and not into y_j is $2y_i - x_e$. Similarly, the sum of edges going out of y_j and not into y_i is $2y_j - x_e$ (see Figure 3.3). Therefore, for the new capacity we have

$$\begin{aligned} y_{i'} &= \frac{1}{2}\left(2y_i - x_e + 2y_j - x_e\right) \\ &= y_i + y_j - x_e. \end{aligned}$$

We also adjust the edge weights for the newly created node, *i.e.*, the weight of an edge from this node to any other node u in the graph would be the sum of the weights of edges from i and j to u. The following pseudo-code best describes our shrinking algorithm:

3.5 Primitive Comb Inequalities

3.5.1 Primitive Comb Inequalities for the TSP

To improve the quality of fractional solution to the PCTSP, valid inequalities can be introduced. Primitive comb inequalities are a class of valid inequalities. In this section, we study these valid inequalities for our version of the PCTSP. Later in Section 3.7, we investigate how effective the primitive comb inequalities are in improving the quality of fractional solutions to the PCTSP.

Algorithm 1 The Shrinking Heuristic

Data: Graph $G = (S', E')$, Edge weights $w : E' \to \mathbb{R}^{\geqslant 0}$, and vertex values $y : S' \to \mathbb{R}^{\geqslant 0}$
Result: The cut V that maximizes $\sum_{e \in E(V, S' \setminus V)} x_e - \max_{k \in V}\{y_k\}$
Initialize each node to be a separate component and set $m_i \leftarrow y_i$, $\forall i \in S'$
for *every edge* $e = (i, j)$ *in* E' **do**
  if $(x_e > y_i - \max\{0, m_i - m_j\})$ ***and*** $(x_e > y_j - \max\{0, m_j - m_i\})$ **then**
    Merge nodes i and j into a new node ℓ
    for *every node* u *in* S' **do**
      if u *is adjacent to* i *or* j **then**
        Let e' denote the edge (ℓ, u)
        $x_{e'} \leftarrow x_{(u,i)} + x_{(u,j)}$
      end
    end
    $y_\ell \leftarrow y_i + y_j - x_e$
    $m_\ell \leftarrow \max\{m_i, m_j\}$
  end
end

Formulation

Comb inequalities are valid inequalities that have been introduced successfully to obtain LP solutions with reduced integrality gaps for the TSP. In addition to this, exact separation algorithms for the comb inequalities have been proposed for the TSP. Because the LPs are solved many times in a branch and bound algorithm to solve the TSP optimally, efficient heuristics have also been proposed for the separation problem which find violated comb inequalities.

In this section, we look at inequalities for the simplest form of such structures known as *primitive combs*. A general comb consists of a set of nodes H known as the *handle*, and a set of t *teeth*, each tooth being a set of nodes that spans out of the handle. A primitive comb restricts each tooth to be a single edge. Primitive comb inequalities have been shown to improve the quality of the fractional solution to the PCTSP.

Let us first look at the primitive comb inequalities for the Traveling Salesman polytope. In what follows, $x(A, B) = \sum_{e=(i,j): i \in A, j \in B} x_e$, where x_e is 1 if the edge e is incorporated in the tour, and x_e is 0 otherwise.

$$x(H, H) + \sum_{j=1}^{t} x(T_j, T_j) \leqslant |H| + \frac{t-1}{2}, \tag{3.5.1}$$

where H, the *handle*, and T_j for $j = 1, 2, \ldots, t$, the *teeth*, are sets of nodes satisfying the following conditions [56]:

$$|H \cap T_j| = |T_j \setminus H| = 1 \qquad j = 1, \ldots, t, \text{with } t \geqslant 3 \, odd \tag{3.5.2}$$

$$T_i \cap T_j = \emptyset \qquad i, j = 1, \ldots, t \tag{3.5.3}$$

Padberg and Hong [93], among others, provide a heuristic separation algorithm for primitive comb inequalities for the TSP. We refer to this as the *odd-component heuristic*.

Odd-Component Heuristic for TSP

In this heuristic, given an optimal LP solution x^* to a TSP instance over a complete graph (V, E), the graph $G^*_{1/2}$ is constructed. This graph has the vertex set V and edge set $\{e \in E : 0 < x^*_e < 1\}$. Let the resulting graph comprise of q components, whose vertex sets are V_1, V_2, $\ldots$, V_q. For each component $i, 1 \leqslant i \leqslant q$, the heuristic determines the subset of edges T_i in the set $\delta(V_i)$ (the set of edges which cross the set V_i) whose LP values are 1. T_i is thus given by $T_i = \{e \in E : x^*_e = 1 \wedge e \in \delta(V_i)\}$. If the cardinality of this set is odd, then the following simple procedure determines a set that either violates the subtour inequality for TSP, or a primitive comb inequality. If two edges in set T_i share a vertex $v \in V \setminus V_i$, then vertex v is included in the set V_i and the procedure is repeated until the edges in T_i are pairwise disjoint. Such a set V_i violates the subtour inequality if $|T_i| = 1$, and the primitive comb inequality if $|T_i| \geqslant 3$. Note that if two teeth T_i and T_j intersect outside of H, adding the common vertex to H would get rid of exactly two teeth, therefore the parity of the teeth of H is preserved. Therefore if a handle has odd number of teeth (which indicated either a violated GSEC or a violated primitive comb inequality) before the removal of a pair of teeth, it will still have an odd number of teeth afterwards.

Balas [9] shows an equivalent version of the comb inequalities are facet defining for the Prize Collecting Traveling Salesman polytope [9]. The following section provides the details of these comb inequalities for our version of the PCTSP.

3.5.2 Primitive Comb Inequalities for PCTSP

Formulation

Balas shows in [9] that the following inequality defines a facet of the Prize Collecting TS polytope. Therefore Inequality (3.5.1) translates to Inequality (3.5.4) in the case of the Prize Collecting TSP.

$$x(H,\, H) + \sum_{j=1}^{t} x(T_j,\, T_j) + z(H) \leqslant |\, H \,| + \frac{t-1}{2}, \tag{3.5.4}$$

Here, $z(H) = \sum_{v \in H} z_v$, and z_v is 1 if the vertex v is left out of the optimal tour (we need to a pay a penalty for it), and z_v is 0 otherwise. When compared to the formulation of the sub problem (Objective Function 2.2.14 and Constraints 3.2.2–3.2.5), one can write $y_v = 1 - z_v$

for all vertices v. As a result, $z(H) = \sum_{v \in H} z_v = \sum_{v \in H}(1 - y_v) = |\, H \,| - y(H)$. Therefore, Inequality (3.5.4) can be written as

$$x(H,\, H) + \sum_{j=1}^{t} x(T_j,\, T_j) - y(H) \leqslant \frac{t-1}{2}, \tag{3.5.5}$$

We show below that a connected component heuristic which has been used by Hong [63] and Land [77] can be applied to separate the primitive comb inequalities introduced by Balas for the PCTSP.

3.5.3 Odd-Component Heuristic for PCTSP

The odd-component heuristic [93] works with an equivalent formulation of the primitive comb inequalities for the Traveling Salesman polytope:

$$x(\delta(H)) + \sum_{j=1}^{t} x(\delta(T_j)) \geqslant 3t + 1. \tag{3.5.6}$$

To be able to use the same heuristic, we first translate Balas's inequality into a similar form:

$$x(\delta(H)) + \sum_{j=1}^{t} x(\delta(T_j)) \geqslant 3t + 1 - 2\sum_{j=1}^{t} z(T_j).$$

Lemma 2. *The following inequality defines a facet of the Prize Collecting TS polytope.*

$$x(\delta(H)) + \sum_{j=1}^{t} x(\delta(T_j)) \geqslant 3t + 1 - 2\sum_{j=1}^{t} z(T_j).$$

Proof. By [9], we know that Inequality (3.5.4) is facet defining. Also, from Constraint 3.2.2, the fractional degree of each node v is $2 \cdot y_v = 2 - 2 \cdot z_v$. Therefore, we can write

$$2\; x(H,\, H) + x(\delta(H)) = \sum_{v \in H} deg(v) = 2\,|\, H \,| - 2\; z(H).$$

Thus,

$$x(H,\, H) = |\, H \,| - z(H) - \frac{1}{2}\; x(\delta(H)).$$

Similarly, we have that

$$2\; x(T_j,\, T_j) + x(\delta(T_j)) = \sum_{v \in T_j} deg(v) = 4 - 2\; z(T_j),$$

for $j = 1, \ldots, t$, which yields

$$x(T_j, T_j) = 2 - z(T_j) - \frac{1}{2}\, x(\delta(T_j)).$$

Therefore,

$$\begin{aligned} & x(H, H) + \sum_{j=1}^{t} x(T_j, T_j) + z(H) = \\ & |\, H \,| - z(H) - \frac{1}{2}\, x(\delta(H)) + \sum_{j=1}^{t} 2 - \sum_{j=1}^{t} z(T_j) - \frac{1}{2} \sum_{j=1}^{t} x(\delta(T_j)) + z(H) = \\ & |\, H \,| - \frac{1}{2}\, x(\delta(H)) + 2t - \sum_{j=1}^{t} z(T_j) - \frac{1}{2} \sum_{j=1}^{t} x(\delta(T_j)) \end{aligned} \tag{3.5.7}$$

From Equation (3.5.7) and Inequality (3.5.4),

$$\frac{1}{2}\, x(\delta(H)) + \frac{1}{2} \sum_{j=1}^{t} x(\delta(T_j)) \geqslant 2t - \sum_{j=1}^{t} z(T_j) - \frac{t-1}{2}.$$

Thus,

$$x(\delta(H)) + \sum_{j=1}^{t} x(\delta(T_j)) \geqslant 3t + 1 - 2 \sum_{j=1}^{t} z(T_j).$$

□

We show below why the odd-component heuristic used by Padberg and Hong to separate primitive comb inequalities for the TSP can be used to separate primitive comb inequalities for the PCTSP without any modifications.

Assume that after running the odd-component heuristic on $G^*_{1/2}$, a component V_i has an odd number of teeth. We remove all non-disjoint teeth and add their common vertex to V_i. Let the handle H be V_i. We assume H has $t \geqslant 3$ teeth as we are more interested in finding violated primitive comb inequalities rather than violated GSECs. By the structure of $G^*_{1/2}$, all the outgoing edges of H have weight exactly equal to 1. Thus, $x(\delta(H)) = t$. Consider a tooth of H, say $T_k = (u_k, v_k)$ (See Figure 3.4). Because of the degree constraints, and the fact that the edge (u_k, v_k) is taking a capacity 1 from each of the two endpoints, we can write $x(\delta(T_k)) = 2y_{u_k} - 1 + 2y_{v_k} - 1 = 2(1 - z_{u_k}) - 1 + 2(1 - z_{v_k}) - 1 = 2 - 2(z_{u_k} + z_{v_k})$. Summing over all t edges and adding $x(\delta(H))$ we get:

$$x(\delta(H)) + \sum_{j=1}^{t} x(\delta(T_j)) = 3t - 2\sum_{j=1}^{t} z(T_j) < 3t - 2\sum_{j=1}^{t} z(T_j) + 1$$

This is exactly 1 short of satisfying the corresponding primitive comb inequality, therefore any component of $G^*_{1/2}$ with odd number of teeth would correspond to a violated inequality that we can introduce as a cut.

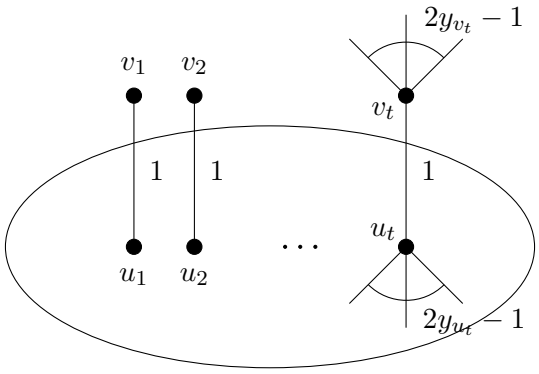


Figure 3.4: A component of $G^*_{1/2}$.

In Section 3.7, we empirically analyze the performance of this heuristic alongside with the shrinking heuristic for GSECs.

3.6 Speeding up the Algorithm

The performance of any branch-and-cut algorithm greatly depends on the measures it takes to prune the search tree. The more nodes the algorithm can avoid, the faster it can arrive at the optimal solution of the problem. An effective way of pruning a large portion of the tree is to achieve bounds on the solution as fast as possible. Consider a maximization problem for instance. As we traverse from the root of the branching tree to the leaves, we introduce more constraints on the linear programming formulation. Therefore, any node has a higher objective value than it's descendants since the child nodes are the more restricted versions of their parents. In this scenario, if we generate lower bounds on the optimal solution quickly, we can ignore any node that has a smaller objective value than the bound. We discuss two efficient ways of making useful lower bounds in this section. The first method is by faster traversal of the tree from the root node to a leaf and the second one is by improving the fractional solutions to LP relaxations of the intermediate nodes using a local search technique.

3.6.1 The Branching Strategy

Generation of applicable bounds in a branch-and-cut algorithm greatly depends on the branching strategy. To speed up the process, we should choose strategies that traverse the tree down to the leaves fairly quickly. A branching strategy determines:

(1) How to choose the next node in the traversal.

(2) Which variable to branch on.

As mentioned above, a fundamental intuition in the branching mechanism is to enable the algorithm to reach the leaves as soon as possible. Since arriving at a node means obtaining an integral feasible solution, such a solution can readily be used as bound for pruning other nodes from the tree, hence speeding up the algorithm. Therefore, we avoid any method that favors traversing the breadth of the tree first, and implement a depth-first mechanism. This strategy treats the list of problems L as a stack. We push the new nodes P_0 and P_1 created via branching to the front of L, and pop the next node to be considered from the front as well.

The choice of the variable for branching is a key factor in how balanced the tree is and how many levels of the tree the branch-and-cut algorithm needs to see to find the optimal. Some of the leading publications on similar problems suggest that in the prize collecting problems such as ours, the decision variables representing the vertices should always have priority for branching compared to the decision variables corresponding to the edges [48]. Our experiments in this research also confirm this intuition. We implemented and compared three branching heuristics in this thesis to verify this.

- Random Branch: Randomly pick a variable, either x_e, for some $e \in E$ or y_v for a $v \in S$.

- Greedy Branch: First, consider y_v for all $v \in S$. Choose a variable y_t that minimizes $|\, y_k - 0.5 \,|$. If all the y variables are integral, then do the same with x_e for all $e \in E$.

- Smart Branch [48]: Consider y_v for all $v \in S$ first. Identify two variables $y' = \max_{y_k \leqslant 0.5}\{y_k\}$ and $y'' = \min_{y_k \geqslant 0.5}\{y_k\}$. In the set $\{y_k : y'/2 \leqslant y_k \leqslant 0.5 + y''/2\}$ choose a variable y_k with the maximum prize β_k. If y_k is integral for every $k \in S$, then do the same with x_e for all $e \in E$ and choose a variables with the highest cost c_e.

The first heuristic is a naïve way of choosing a variable for branching, and is implemented for benchmarking purposes. The second heuristic gives priority to y variables for branching, but only considers closeness to 0.5 as the criteria for branching. The reasoning behind it is that the variables closer to 0 or 1 are more likely to be set to their closest integer in the following nodes in the tree. Therefore, the half-integral variables are deemed to be the critical ones in the branching process. Finally, the last heuristic, while also giving priority to y variables, considers the coefficients as well. As experimental evidences show, the higher the prize of a vertex (or the cost of an edge), the sooner it should be selected for branching.

3.6.2 The Local Search

Another way to derive a bound in the branching process is by "rounding" the fractional solution of an LP relaxation at each node. By rounding, we mean obtaining an integral

solution based on the fractional values that the LP solution has assigned to the variables. If done properly, rounding can improve the incumbent by finding a better integral solution, thus resulting in a new bound.

We should note that running a rounding algorithm at every node in the branching tree comes with its computational cost. For this reason, we should make a trade-off between the quality of the bound that we obtain and the running time that the algorithm needs to round the variables. A sophisticated heuristic that produces relatively high lower bounds and takes a long time for doing so might only slow down the branch-and-cut algorithm. For this reason, we choose simpler local search heuristics with light computational costs that can produce reasonably good solutions. We present the local search in Algorithm 2.

The local search starts with a tour T that only includes the depot r. Along the way, it makes use of the following methods:

- oneOpt: Insert a new node v to the existing tour T in a location that maximizes the net benefit. Assume the tour is given by $u_0, u_1, \ldots, u_k, u_0$ for some k, $1 \leqslant k \leqslant n-1$ where $u_0 = r$. oneOpt finds a vertex v such that $v \notin \{u_0, u_1, \ldots, u_k\}$ and detects the best location to insert v, *i.e.*, indices i and $i+1$, $i < k$, for which the net benefit $\beta_v + c_{u_i, u_{i+1}} - c_{u_i, v} - c_{v, u_{i+1}}$ is maximized. It then returns the new tour $u_0, u_1, \ldots, u_i, v, u_{i+1}, \ldots, u_k, u_0$. The new tour collects the prize for v, β_v, and does not pay the cost $c_{u_i, u_{i+1}}$ for the former edge between nodes u_i and u_{i+1}. Instead, it has to pay the extra costs $c_{u_i, v}$ and $c_{v, u_{i+1}}$ for the new edges connecting v to the rest of T.

- twoOpt[1]: Insert two new nodes v_1 and v_2 into some locations in an existing tour T and remove a node u_t, $1 \leqslant t \leqslant k$ from T to increase the net benefit. Assume the tour is given by the nodes $u_0, u_1, \ldots, u_k, u_0$ for some k, $1 \leqslant k \leqslant n-1$ where $u_0 = r$. twoOpt finds v_1 and v_2 as well as insertion locations for them in T, and an existing node u_t to remove from the tour, hence increasing the size of the tour by one. Assume v_1 is to be inserted between the nodes u_i and u_{i+1} and v_2 is to be inserted between the nodes u_j and u_{j+1}. It does it in such a way that the benefit $\beta_{v_1} + \beta_{v_2} + c_{u_i, u_{i+1}} + c_{u_j, u_{j+1}} + c_{u_{t-1}, u_t} + c_{u_t, u_{t+1}}$ minus the cost $\beta_{u_t} + c_{u_i, v_1} + c_{v_1, u_{i+1}} + c_{u_j, v_2} + c_{v_2, u_{j+1}} + c_{u_{t-1}, u_{t+1}}$ is maximized. Note that we make a benefit from the prizes of nodes v_1 and v_2, and from not including the edges between former neighbours in the tour u_i and u_{i+1}, u_j and u_{j+1}, u_{t-1} and u_t, and finally u_t and u_{t+1}. We also incur a cost from loosing the benefit of node u_t as well as from having to include new edges between u_i and v_1, v_1 and u_{i+1}, u_j and v_2, v_2 and u_{j+1}, and finally u_{t-1} and u_{t+1}.

- oneForce: At the beginning of the local search, the method oneOpt may fail to add a node to the tour T. This is the cases when the costs of adding any node to the graph

[1]This method should not be confused with the well-known 2-opt local search heuristic of Croes that removes the crossovers in a tour.

exceed the benefits. The trivial solution is a tour containing only the depot in such scenarios. We reject single-vertex tours as possible solutions to the problem. To avoid such solutions, we use oneForce to push at least one extra vertex to the tour T. It adds a vertex v into the tour $T = r$ such that the net benefit $\beta_v - 2 \times c_{r,v}$ is maximized. Then, it returns the new tour r, v, r.

- oneSwap: given the tour T, it checks if removing one of the vertices of T and adding a new vertex to it (to potentially a new location) can improve the objective function. If so, it returns the new tour. Otherwise, it returns the old tour T. Assume T is given as $u_0, u_1, \ldots, u_k, u_0$ for some $1 \leqslant k \leqslant n-1$ where $u_0 = r$. If the node v is to be inserted between the nodes u_i and u_{i+1}, and node u_t is to be removed, then the net benefit would be $\beta_v + c_{u_{t-1},u_t} + c_{u_t,u_{t+1}} + c_{u_i,u_{i+1}} - \beta_{u_t} - c_{u_i,v} - c_{v,u_{i+1}} - c_{u_{t-1},u_{t+1}}$. Note that we benefit from the prize of node v as well as from avoiding the costs of former edges (u_{t-1}, u_t), (u_t, u_{t+1}), and finally (u_i, u_{i+1}). We also incur a cost by loosing the benefit of node u_t and also by the cost of the new edges (u_i, u_v) and (u_v, u_{i+1}) that connect v to the rest of the tour as well as the new edge between u_{t-1} and u_{t+1}.

Algorithm 2 Local Search Algorithm

Data: A graph $G = (S, E)$, the depot $r \in S$, set of costs c_e, and a set of prizes βj.
Result: returns a tour T that covers the depot r

```
finished ← FALSE
T ← {r}
while not finished do
    T ← oneOpt(G, c, β)
    if size(T) ⩾ 2 then
        T ← twoOpt(G, c, β)
    end
    if no changes to T then
        if T == {r} then
            T ← oneForce(G, c, β)
        end
        else
            finished ← TRUE
        end
    end
end
finished ← size(T) ⩽ 2
while not finished do
    T ← oneSwap(G, c, β)
    if no changes to T then
        finished ← TRUE
    end
end
```

In the beginning, we use the local search on the entire set S to find the first incumbent. Also, each time we solve the LP relaxation for a node, we run the local search on the set of nodes $\{v_k : y_k \geqslant 0.3\}$ to find an integral solution that is guided by the LP solution. This solution may have a better objective value than that of the incumbent. In this case, we update the incumbent which automatically updates the lower bound for the pruning as well.

3.7 Computational Results

We have developed a software, implementing the branch-and-cut framework. In this section, we report on computational results from running our software on a standard library of TSP and VRP instances. First, we report the efficacy of the two adapted heuristics for GSECs and Primitive Comb Inequalities and analyze the behavior of the corresponding separations. The computational results show how effective the heuristic algorithms are in cutting off non-integral solutions of the LP. In this section, we are only solving the LP formulation of the problem that corresponds to the root node of the branching tree. In the next set of computational results, we run the entire branch-and-cut algorithm on various instances. In this set, we compare the branching heuristics Greedy Branch and Smart Branch against one another using two different depth-first traversals of the branching tree. We also compare the running time of the runs against CPLEX ILP solver which uses a generic branch-and-bound algorithm. The software has been implemented in C++, and the LP was solved using CPLEX 12.5. The code was run on an Intel 2.8 GHz processor with a maximum time limit of 14,400 seconds (4 hours). We report the results after the period of 4 hours if the execution has terminated, or report failure otherwise.

3.7.1 Instances

We have transformed a total of 42 instances out of the TSPLIB [95] library for our experiments. Out of the 42 instances, 10 were VRP instances and 32 symmetric TSP instances. In the VRP instances, each node is associated with a demand value. We use this demand as the prize of covering that node. For the TSP instances, which do not come with demands, we merely generate node prizes independently and uniformly at random in the range of 1 to the maximum cost of that instance. The reason for instance-specific generation of the prizes reason is to ensure that neither of the cost or prize values can dominate the other. Note that if the prizes are significantly larger than costs, the problem reduces to a TSP, and if the cost values dominate the prize values, the trivial solution of only picking the depot would be optimal.

3.7.2 Separation Heuristics

We first describe the instance that we have used. Then we discuss the experiments we designed to test the behaviour and performance of the two heuristics, namely the shrinking heuristic for separation of the GSECs and the odd-component heuristic for separation of the primitive comb inequalities. We next explain how we compared these results and analyze their relative gap with respect to the optimal integral solution whenever such a solution is available. We summarize the results in Tables 3.1 and 3.2.

Designing the Experiments

We analyze the adapted heuristics based on three difference aspects:

(1) The solution quality.

(2) The number of cuts (violated constraints) they detect.

(3) The running time.

The major factor in the success of a heuristic is indeed if it can produce near optimal solutions in a timely manner. Nevertheless, the number of cuts a heuristic can generate can give us insight about how well that heuristic is adapted for a certain type of polytope. If near optimality can be achieved with fewer cuts, we have some empirical indications that the heuristic is well-suited for the polytope of the given problem. For this reason, we also report the number of cuts for both the heuristics, as well as for an optimal procedure that uses LP for separation of GSECs. In what follows, we explain the method used for making the comparisons.

Shrinking Heuristic. To evaluate the efficacy of the shrinking heuristic, we compare them against a procedure that uses linear programming for separation of the GSECs, that is, a formulation that solves a cut-set equivalent of (3.4.5)–(3.4.7) for each node to find the violated GSECs. Three parameters mentioned above are reported for the heuristic GSEC separation, as well as the LP separation.

Odd-component Heuristic. To evaluate this heuristic, we first solve instances with the GSEC heuristic only, and then use odd-component heuristic to find violated primitive comb constraints. This way, we can measure how the solution quality may improve (and how many new cuts are introduced), and weigh it against the extra time we need to spend for running the second heuristic. The improvements are also reported in the right-most column of Table 3.2. Also, using the Odd-Component heuristic on top of the shrinking heuristic, we may get a few extra cuts for some instances. The number of such cuts can be seen in the third column of Table 3.2. To gain insight as how we compare against an optimal integral solution, we solve the ILP version of the problem using the built-in branch-and-bound method of CPLEX alongside regular Subtour Elimination Constraints (SECs). Note that

to separate SEC inequalities, all we need to do is to find an integral subtour in a solution, which can be done in polynomial time using a connected component algorithm. This method can be slow for larger instances, therefore in many cases we have not been able to solve the ILP optimally in the time limit of 4 hours. For such instances, the gap percentages has been left blank. Also, in Table 3.1, "t.l." stands for *too long*, and is used for running times that are greater than 14,400 seconds.

It is natural to assume the heuristics would be most beneficial for large enough instances. As Tables 3.1 and 3.2 show, there is not much room for improvement for small instances as usually even the linear programming formulation can come up with an integral solutions without the aid of GSEC or primitive comb inequality constraints.

As soon as the size of the instance goes beyond 50 points, the time spent on the LP separation shows a huge increase while the cost of heuristic separation is still relatively low. This is not so surprising as the LP separation has to solve an entire linear programming problem for each node of the graph. For small instances, the overhead caused by a few extra problems is negligible, but for larger instances it can be prohibitive. Also, for the instances for which an integral solution is available, the gap between the two heuristic solutions combined and the optimal solution is very small. In most cases the gap is very close to 0 and only for one instance it goes as high as 2%. In larger instances, both heuristics perform consistently well as they introduce many cuts to the formulation and come up with reasonable solutions while the LP separation of GSECs and the ILP formulation do not return with a solution in the period of 4 hours.

A comparison between using GSEC heuristic separation alone and using both heuristics together reveals that although there are cases for which the solution quality can benefit significantly from running both heuristics instead of one, the GSEC separation seems to be promising on its own in many cases. The relatively long extra time spent on the separation of primitive combs results in the introduction of a few extra cuts indeed. However, the improvement in the solution quality is not very significant. This is perhaps to be expected since we run the primitive comb separation after all the GSEC cuts are introduced. As a result, the LP has already closed the gap between the fractional and the optimal integral objective value, and therefore, improvements at this point come in very small portions. In the next set of results, we show that by using different sequences of running the heuristics, one can speed up the running time and get the best of the two heuristics.

3.7.3 The Branching Mechanism

Now, we wish to find the exact solutions of the instances. To do so, we use a mix of different branching heuristics and traversal rules and draw comparison between the running time of each method. As mentioned earlier, we also use CPLEX ILP solver as a benchmark.

	LP GSEC Sep.			Heuristic GSEC Sep.		
Instance	**Obj.**	**Cuts**	**Time**	**Obj.**	**Cuts**	**Time**
eil13	3228.61	0	<1	3228.61	0	<1
ulysses22	-1964.93	16	1	-1964.93	13	<1
bayg29	-44.35	33	4	-44.35	14	<1
eil33	-2924.76	44	7	-2924.76	18	<1
att48	1358.75	0	7	1358.75	0	<1
hk48	551.65	1	7	551.65	1	<1
eil51	-4454.6	2	32	-4454.6	2	<1
st70	-6433.23	94	423	-6433.23	26	1
eilB76	-7442.17	28	964	-7442.31	11	3
eilC76	-6898.96	27	354	-6900.36	7	1
eilD76	-6774.68	33	434	-6774.74	15	2
pr76	5568.42	0	50	5568.42	0	<1
gr96	-9531.81	83	2740	-9531.81	26	2
rat99	-9260.24	80	4802	-9260.87	15	14
rd100	-10119.1	189	2501	-10119.1	35	2
eil101	-8988.51	73	1108	-8988.51	17	5
eilA101	-9160.86	60	3153	-9160.86	17	1
eilB101	-9710.61	61	1107	-9710.61	18	2
lin105		N/A	t.l.	-87.93	354	4371
pr107	339.84	0	238	339.84	0	<1
gr120	-10451.1	85	3367	-10451.1	31	80
gr137	-13070.5	112	4306	-13070.5	26	19
pr144	532.69	11	4677	532.69	6	<1
ch150		N/A	t.l.	-8891.09	75	68
pr152	984.54	13	10780	984.54	13	2
rat195		N/A	t.l.	-18301.7	39	972
d198		N/A	t.l.	-19183.9	31	3
korB200		N/A	t.l.	-138.68	111	7
gr202		N/A	t.l.	-19859.1	48	70
ts225	1296.11	0	8466	1296.11	0	14
gr229		N/A	t.l.	-20520.4	67	8
gil262		N/A	t.l.	-24784.5	96	27
a280		N/A	t.l.	-24933.6	60	67
lin318		N/A	t.l.	-1883.21	365	1906
fl417		N/A	t.l.	-41406.8	48	6
gr431		N/A	t.l.	-41751	123	823
pr439		N/A	t.l.	322.41	20	70
pcb442		N/A	t.l.	-43489.5	142	338
d493		N/A	t.l.	-48934.3	43	25
p654		N/A	t.l.	-65478.8	71	1068
d657		N/A	t.l.	-64671.9	125	1162
gr666		N/A	t.l.	-65273.1	169	54

Table 3.1: Computational results for the GSEC heuristic separation

	Heuristic GSEC + Primitive Comb				
Instance	**Obj.**	**Extra Cuts**	**Time**	**Gap %**	**Improv.**
eil13	3228.61	0	<1	0	0
ulysses22	-1964.93	5	<1	0	0
bayg29	-44.35	0	1	0	0
eil33	-2924.76	0	1	0	0
att48	1358.75	0	1	0	0
hk48	551.65	0	1	2.01	0
eil51	-4451.36	10	1	0.03	3.24
st70	-6431.62	6	2	0	1.61
eilB76	-7441.27	4	4	0	1.04
eilC76	-6900.17	6	1	0.03	0.19
eilD76	-6773.72	7	7	0	1.02
pr76	5568.42	0	1	0	0
gr96	-9529.7	36	5	0.02	2.11
rat99	-9260.32	4	21	0.04	0.55
rd100	-10119.1	20	3	0	0
eil101	-8988.02	4	6	0.03	0.49
eilA101	-9160.59	2	2	0.01	0.27
eilB101	-9710.34	2	7	0	0.27
lin105	-87.93	0	5565		0
pr107	339.84	0	<1	0	0
gr120	-10451.1	2	185	0	0
gr137	-13068.3	10	31	0.01	2.2
pr144	532.69	0	<1		0
ch150	-8874.46	8	85	0.53	16.63
pr152	984.54	0	2		0
rat195	-18296.1	119	6567	0.09	5.6
d198	-19183.9	5	6	0	0
korB200	-138.68	0	8		0
gr202	-19858.2	38	281	0	0.9
ts225	1296.11	0	20	0	0
gr229	-20516.7	16	13	0.02	3.7
gil262	-24777	32	49	0.04	7.5
a280	-24921.1	49	122	0.02	12.5
lin318	-1842.91	94	4420		40.3
fl417	-41406.8	40	8		0
gr431	-41748.6	38	2041	0.01	2.4
pr439	322.41	0	108		0
pcb442	-43489.5	20	963		0
d493	-48934.3	39	38		0
p654	-65478.8	35	1317		0
d657	-64671.9	21	1482		0
gr666	-65266	80	113	0.03	7.1

Table 3.2: Computational results for the Primitive Comb Inequalities heuristic separation

			Fixed			Deterministic		
Instance	**Obj.**	**ILP Time**	**Optimal**	**Time**	**Gap %**	**Optimal**	**Time**	**Gap %**
eil13	3228.61	1	✓	<1		✓	<1	
ulysses22	-1964.92	2	✓	<1		✓	<1	
bayg29	-44.35	2	✓	<1		✓	<1	
eil33	-2924.76	2	✓	<1		✓	<1	
att48	1358.75	1	✓	<1		✓	<1	
hk48	562.98	<1	✓	<1		✓	<1	
eil51	-4450.18	2	✓	<1		✓	<1	
st70	-6432.24	3	✓	4		✓	<1	
eilB76	-7441.65	1	✓	1		✓	2	
eilC76	-6898.31	2	✓	7		✓	131	
eilD76	-6774.3	2	✓	8		✓	8	
pr76	5568.42	<1	✓	<1		✓	1	
gr96	-9527.97	6	✓	46		✓	262	
rat99	-9256.63	6	✓	223		✓	253	
rd100	-10119.1	11	✓	2271		✓	659	
eil101	-8986.01	6	✓	56		✓	156	
eilA101	-9160.25	3	✓	5		✓	2	
eilB101	-9710.79	3	✓	8		✓	263	
lin105	-72	t.l.	✓	1860		✓	1717	
pr107	339.84	<1	✓	<1		✓	<1	
gr120	-10452.1	24	✓	179		✓	180	
gr137	-13067.1	59	✓	472		✓	7061	
pr144	>390.19	t.l.	-	t.l.	26.7%	-	t.l.	26.7%
ch150	-8827.78	14226	✓	1166		✓	2158	
pr152	984.54	t.l.	✓	12		✓	7	
rat195	-18279.6	2462	-	t.l.	0.3%	-	t.l.	6.15%
d198	-19183.3	95	✓	1769		-	t.l.	0.11%
kroB200	-138.68	t.l.	✓	17		✓	13	
gr202	-19859.1	32	✓	9862		✓	6762	
ts225	1296.11	1	✓	<1		✓	<1	
gr229	-20515.7	887	✓	805		✓	8586	
gil262	-24767.6	545	✓	3205		-	t.l.	1.9%
a280	-24878.5	t.l.	✓	3520		✓	10409	
lin318	>-7617.46	t.l.	-	t.l.	N/A	-	t.l.	N/A
fl417	>-41424	t.l.	-	t.l.	0.06%	-	t.l.	0.06%
gr431	-41742.6	5180	✓	2651		-	t.l.	0.45%
pr439	168.6	t.l.	✓	17		✓	24	
pcb442	>-43514.9	t.l.	-	t.l.	0.07%	-	t.l.	0.08%
d493	>-48946.8	t.l.	-	t.l.	0.02%	-	t.l.	0.03%
p654	>-65493	t.l.	-	t.l.	0.02%	-	t.l.	0.02%
d657	>-64690.2	t.l.	-	t.l.	0.03%	-	t.l.	0.03%
gr666	>-65381.8	t.l.	-	t.l.	0.79%	-	t.l.	0.79%

Table 3.3: Computational results for the Smart Branch

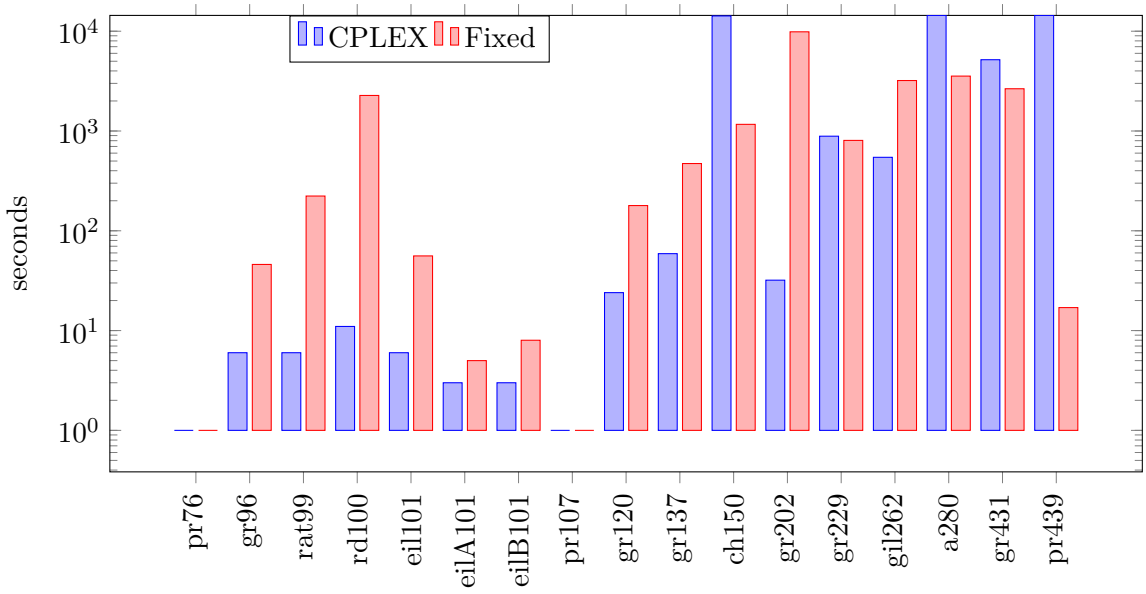


Figure 3.5: The comparison between the running time of CPLEX and the Branch-and-Cut algorithm for SMART BRANCH heuristic with fixed branching

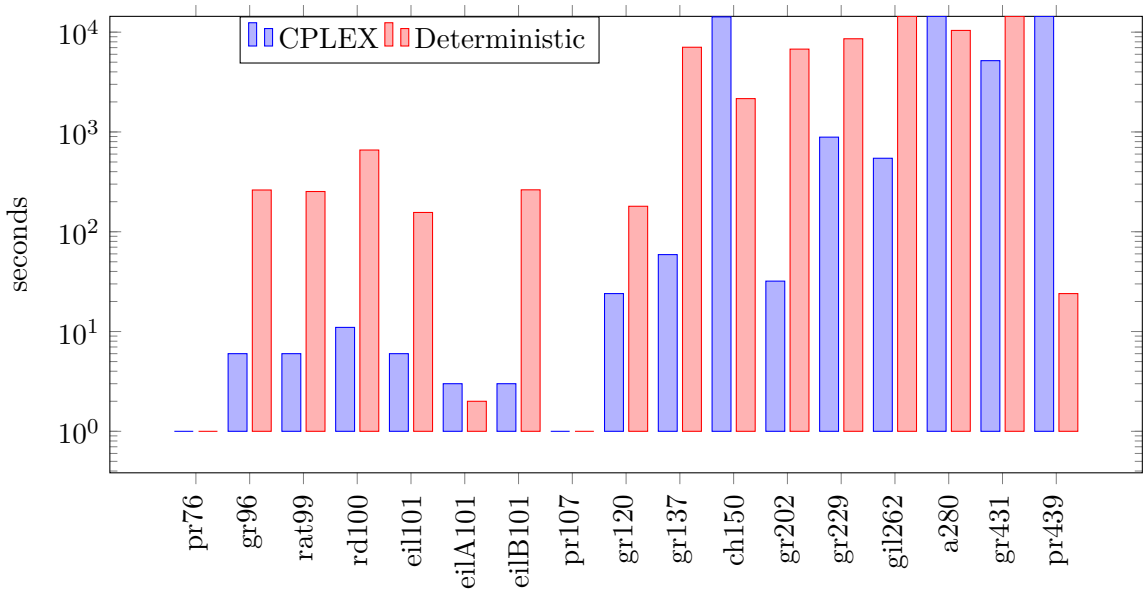


Figure 3.6: The comparison between the running time of CPLEX and the Branch-and-Cut algorithm for SMART BRANCH heuristic with deterministic branching

			Fixed			Deterministic		
Instance	**Obj.**	**ILP Time**	**Optimal**	**Time**	**Gap %**	**Optimal**	**Time**	**Gap %**
eil13	3228.61	1	✓	<1		✓	<1	
ulysses22	-1964.92	2	✓	<1		✓	<1	
bayg29	-44.35	2	✓	1		✓	1	
eil33	-2924.76	2	✓	<1		✓	<1	
att48	1358.75	1	✓	<1		✓	<1	
hk48	562.98	<1	✓	<1		✓	<1	
eil51	-4450.18	2	✓	1		✓	<1	
st70	-6432.24	3	✓	5		✓	<1	
eilB76	-7441.65	1	✓	1		✓	<1	
eilC76	-6898.31	2	✓	6		✓	5	
eilD76	-6774.3	2	✓	3		✓	2	
pr76	5568.42	<1	✓	1		✓	<1	
gr96	-9527.97		✓	217		✓	62	
rat99	-9256.63	6	✓	36		✓	61	
rd100	-10119.1	11	✓	287		✓	392	
eil101	-8986.01	6	✓	45		✓	12	
eilA101	-9160.25	3	✓	<1		✓	3	
eilB101	-9710.79	3	✓	6		✓	5	
lin105	-72	t.l.	✓	3086		-	t.l.	<0.01%
pr107	339.84	<1	✓	<1		✓	1	
gr120	-10452.1	24	✓	623		✓	177	
gr137	-13067.1	59	✓	2431		✓	2208	
pr144	>390.19	t.l.	-	t.l.	26.7%	-	t.l.	26.7%
ch150	-8827.78	14226	✓	401		✓	3404	
pr152	984.54	t.l.	✓	1		✓	2	
rat195	-18279.6	2462	✓	8515		-	t.l.	1.74%
d198	-19183.3	95	✓	66		✓	30	
kroB200	-3151.54	t.l.	✓	17		✓	85	
gr202	-19859.1	32	✓	1073		-	t.l.	0.28%
ts225	1296.11	1	✓	3		✓	<1	
gr229	-20515.7	887	✓	1260		✓	639	
gil262	-24767.6	545	✓	3905		-	t.l.	1.18%
a280	-24878.5	t.l.	-	t.l.	1.6%	✓	2386	
lin318	>-7617.46	t.l.	-	t.l.	N/A	-	t.l.	N/A
fl417	>-41424	t.l.	-	t.l.	0.06%	-	t.l.	0.06%
gr431	-41742.6	5180	-	t.l.	0.45%	-	t.l.	0.45%
pr439	168.6	t.l.	✓	2		✓	30	
pcb442	>-43514.9	t.l.	-	t.l.	0.07%	-	t.l.	0.08%
d493	>-48946.8	t.l.	-	t.l.	0.02%	-	t.l.	0.03%
p654	>-65493	t.l.	-	t.l.	0.02%	-	t.l.	0.02%
d657	>-64690.2	t.l.	-	t.l.	0.03%	-	t.l.	0.03%
gr666	>-65381.8	t.l.	-	t.l.	0.24%	-	t.l.	0.79%

Table 3.4: Computational results for the Greedy Branch

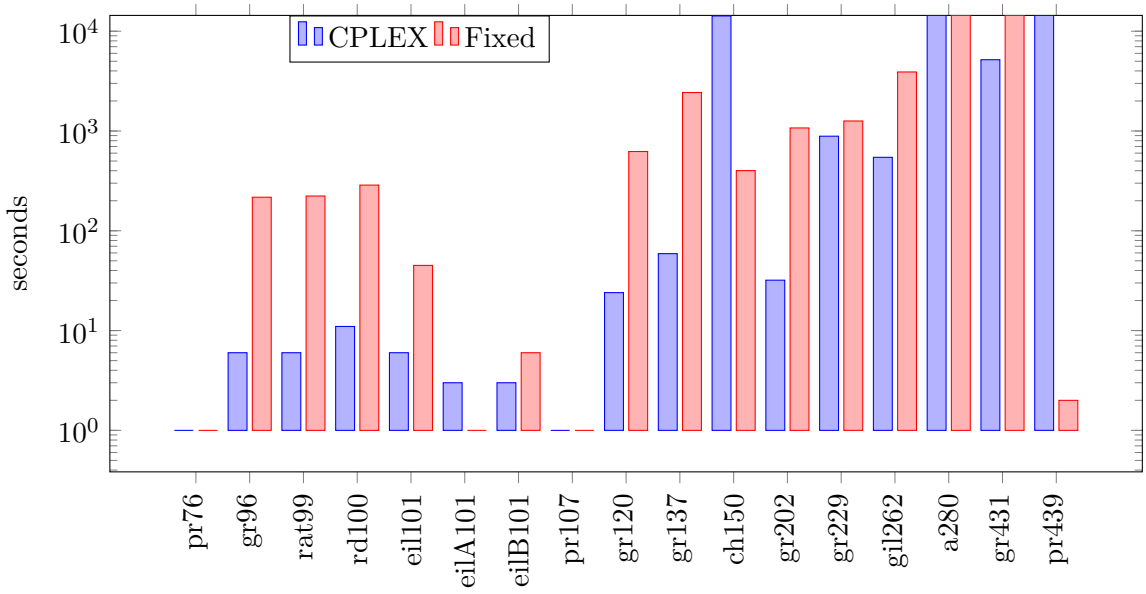


Figure 3.7: The comparison between the running time of CPLEX and the Branch-and-Cut algorithm for Greedy Branch heuristic with fixed branching

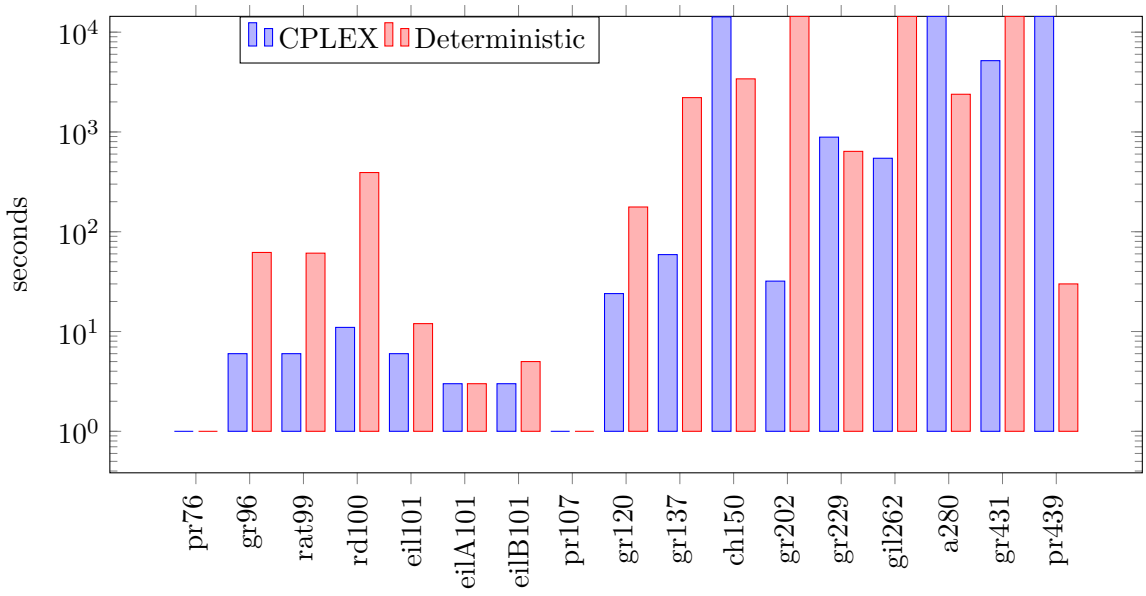


Figure 3.8: The comparison between the running time of CPLEX and the Branch-and-Cut algorithm for Greedy Branch heuristic with deterministic branching

Designing the Experiments

We compare two different branching heuristics:

(1) Greedy Branch

(2) Smart Branch

We have also implemented a random branching method, Random Branch. As expected, basic experiments with this branching method showed a very poor performance. Therefore, we do not include this method in the computational results of this section. We also use two different depth-first traversals of the branching tree:

(1) Fixed: In the first one, we always visit the node corresponding to P_0 first and recursively traverse the entire subtree. Then we move to P_1 and its corresponding subtree. Remember that P_0 and P_1 are the two problems creating when fixing a variable to 0 and 1 respectively.

(2) Deterministic: In the second method of traversal, we visit the nodes based on the value of the variable on which we are branching. Assume x denotes this variable. In this method, we first recursively visit the subtree of P_0 whenever $x \leqslant 0.5$, and we visit P_1's subtree first whenever $x > 0.5$.

As a result, we have a total of 5 different runs: 4 runs of the branch-and-cut algorithm as described above and one run of the CPLEX ILP solver for comparison. We bring the results in Tables 3.3 and 3.4. The objective value might be achieved through any of the methods we compare in the tables. We mention it in the second column whenever it is available. As before, "t.l." stands for too long and indicates cases where the algorithm could not produce the optimal in the 4 hour time window. When that happens, we compare the best incumbent against the best lower bound of the algorithm and report in the "Gap %" field.

For most of the cases, Fixed Smart Branch and Deterministic Smart Branch either solve the instance to optimality or produce a very close solution, with the former failing to find the optimum in only 9 instances. To our surprise, the Fixed Smart Branch outperforms the Deterministic Smart Branch on many instances. Closer investigations reveal that near the end of the branching procedure, giving priority to P_0 against P_1 results in fewer infeasible nodes. The branching algorithm generates infeasible nodes when some of the cuts already added to the node chop off the value 1 for a particular variable, but nonetheless, the algorithm tries an assignment of 1 to it. For instance, if a variable y_i has a value 0.7 in the LP solution while $yi = 1$ is infeasible due to cuts, the Fixed Smart Branch quickly resolves the matter by setting y_i to 0 while the Deterministic Smart Branch tries fixing y_i to 1 first. This means that the Smart Branch favors the variables

that should be set to 0 near the end of the traversal. A bar chart comparison between the performance of the Smart Branch using the two traversals against CPLEX ILP solver is presented in Figures 3.5 and 3.6. In these two figures, the y-axis represents the running time in seconds for both algorithms. Also note that the scale of y-axis is logarithmic for better comparison. A bar that breaks the horizontal barrier on the top means the algorithm needed more than 4 hours to optimize the instance.

As for the Fixed Greedy Branch and Deterministic Greedy Branch, the number of successful runs (those that find the optimal) is almost the same as their Smart Branch counterparts, although the failure rate slightly increases. For the Greedy Branch, the performance of the fixed and deterministic traversals is more balanced as in many cases the deterministic outperforms the fixed traversal. As opposed to the Smart Branch, we do not as often face a situation in which the algorithm instantiates infeasible P_1 nodes near the end of the branching procedure. Bar chart comparison between the performance of the Greedy Branch using the two traversals against CPLEX ILP solver is also provided in Figures 3.7 and 3.8.

3.8 Conclusion and Future Work

In this chapter, we presented the general solution framework for solving the Skill Vehicle Routing Problem using a column generation method. The process of solving the SVRP is a complex task that requires intricate interaction between various techniques. To be able to solve the problem in a timely manner for larger instances, we had to break down the framework into different components (or modules). A schematic view of these components is given in Figure 3.9. In this figure, the dark modules are the ones that we have implemented and tested successfully. Two of these modules, namely the separation algorithms and local search, are suggested as topics for further research. We will shortly discuss how we plan to extend these parts in the upcoming projects. As of this thesis, we have not implemented any heuristics for the master problem, although we would like to emphasize the importance of heuristic solutions for SVRP. We will discuss the future work we have planned for this component as well.

Separation Algorithms

We considered two types of cuts in this research: the Generalized Subtour Elimination Constraints and the Primitive Comb Inequalities. For each of them, we have adapted fast separation heuristics from similar algorithms used in the context of TSP. Effective cuts are crucial in the performance of any branch-and-cut algorithm. Ideally, we look for cuts that bring the fractional solution as close as possible to the optimal integral solution so that we do not need to test too many nodes of the branching tree. Inevitably, one should make a compromise between the computational cost of the separation methods associated with cuts

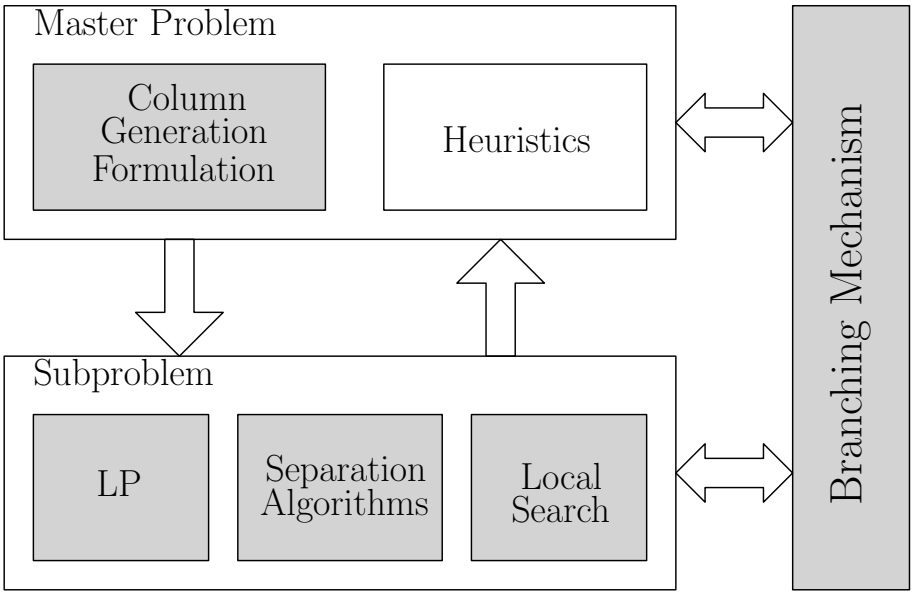


Figure 3.9: A Schematic View of the Solution Framework for SVRP

and the improvement they cause in the fractional solution. Our investigation has shown that the two classes of cuts we have considered so far are good candidates for implementation. Furthermore, we believe other types of cuts should be considered as well as a possible future work. We plan to look into heuristic separations for Path Cover and Cycle Cover Inequalities as the next step.

Local Search for the Subproblem

We implemented a lightweight local search algorithm to both solve the PCTSP and round its fractional solution to a possible incumbent. Similar to the case of separation algorithms, there is a trade-off between the time complexity of any heuristic algorithm and its efficacy. A good feasible solution to the PCTSP can chop off a large portion of the search space. Therefore, we suggest that more sophisticated heuristics than our local search can significantly improve the running time of our algorithm. This local search may also be specialized to round fractional solutions to relatively good feasible ones. We plan to empower and fine tune our local search for that purpose in the short term and propose to investigate a tabu search in the next steps.

Heuristics for the Master Problem

An important possible direction to take would be to consider heuristic algorithms for the SVRP. One of our main intentions for developing exact-solution software for the SVRP was to provide a benchmarking tool for any heuristic algorithm for the problem. In the past two decades, heuristics have earned their place among the most reliable tools for combinatorial

optimization problems, especially against large scale real-world problems. We wish to study heuristic algorithm for the SVRP from two different perspective:

- as independent solution strategies for the SVRP: We have observed that even local searches performed on small neighbourhoods produce satisfactory near-optimal solutions in astonishingly faster than our best efforts with the branching framework. Our exact-solution framework can help in this regard with providing a guideline on what sort of ideas may be successful.

- as help for speeding up the framework: Our solution strategy can greatly benefit from the insight that heuristics can provide to make more educated guesses in its search of the solution space.

Fine-tuning

Finally, the program that we have developed works around a large set of parameters that we have adjusted to the best of our knowledge at the time. These parameters, ranging from the tolerance of degenerate solutions to the number of cuts of each sort that should be allowed in each iteration, have a large impact on the performance. Only through extensive experiments would we be able to adjust the software to perform at its best. We will attend to this matter promptly by going about a systematic fine-tuning of the software.

Part III

Ordered Instances of the Scheduling Problem

Chapter 4

PTAS for Ordered Instances of the Resource Allocation Problems

4.1 Introduction and Problem Definition

In the general resource allocation problem, we are given a bipartite graph $H = (I, P, E)$ where the set of vertices I represents m indivisible resource items (or simply items) to be allocated to the set P of n players (or customers). The sets I and P form the two sets of the bipartition, and are indexed by numbers in $[m]$ and $[n]$ respectively. Thus, $I = \{x_1, x_2, \ldots, x_m\}$ and $P = \{p_1, p_2, \ldots, p_n\}$. Each player $p_j \in P$ has a utility function $val_j(i) = v_i \geqslant 0$ for each item $x_i \in I$. This represents the value of item x_i for player p_j. If item x_i is adjacent to player p_j, then its value for her is $v_i > 0$ (v_i is a positive integer), otherwise its value is zero. This is represented in the graph H via the edge set E. If an item x_i has a value $v_i > 0$ to a player p_j, then there exists an edge $e \in E$ that connects x_i to p_j in H, and we say player p_j is adjacent to item x_i. If there is no edge between an item $x_i \in I$ and a player $p_j \in P$, then the item x_i has a value of zero to player p_j. Allocating an item $x_i \in I$ with value $v_i \in \mathbb{Q}_{>0}$, where $\mathbb{Q}_{>0}$ is the set of positive rational numbers, to a player adjacent to the item increases the utility of that player by v_i. Our goal is to find a feasible allocation of the resource items to the players that optimizes an objective function.

We assume there are no items of degree zero in H. If there exists such an item, we can safely remove it from the graph since it is not accessible to any player. In a feasible allocation (henceforth, *feasible assignment*) for n players, the set I is partitioned into n subsets, where each subset is allocated to the player with the same index. In other words, $I = I^1 \cup I^2 \cup \cdots \cup I^n$, where items in I^j are allocated to player $p_j \in P$ who is adjacent to these items. Such a partition is denoted by $\boldsymbol{\mathcal{A}} = (I^1, I^2, \ldots, I^n)$.

The objective function that is optimized depends on the particular resource allocation problem we solve. In the Max-Min allocation problem, we seek a feasible solution that ensures that each player receives utility *at least* $t > 0$, assuming that we can guess the largest

possible value for the parameter t. In other words, we seek a partition $\boldsymbol{\mathcal{A}} = (I^1, \ldots, I^n)$ such that $\sum_{i:x_i \in I^j} v_i \geqslant t$, $\forall j \in [n]$. A sufficiently good guess of t can be obtained by doing a binary search. It is worth mentioning that the assignments made in the Max-Min allocation problem do not necessarily need to be partitioning of the items, meaning that one can leave some items unassigned. But since the goal is maximize the some of items values assigned, restricting the assignments to cover the entire of set of items will not harm the objective value. Therefore, to avoid further complications, we assume the assignments made are in fact partitions of the entire set of items. On the other hand, in the Min-Max allocation problem, we have item costs rather than item values. For instance, v_i's can be the construed as processing times of a set of jobs and players can be processors, and we are required to assign every job to some processor. Naturally, we wish to allocate jobs to machines such that the overall finish time, or *make-span*, is minimized. Therefore, we seek a feasible solution that ensures each player receives a utility of *at most* $t > 0$ for the smallest t possible. In other words, we seek a partition $\boldsymbol{\mathcal{A}} = (I^1, \ldots, I^n)$ such that $\sum_{i:x_i \in I^j} v_i \leqslant t$, $\forall j \in [n]$.

In this chapter, we study cases where we can obtain PTAS for these two resource allocation problems. We start with cases where the bipartite graph $H = (I, P, E)$ that models the problem is *convex*. A bipartite graph $H = (I, P, E)$ with two set of vertices I and P, is *convex* if there is an ordering $<_I$ of the vertices in I such that the neighbourhood of each vertex in P consists of consecutive vertices, *i.e.*, the neighbourhood of each vertex in P forms an interval. Indeed, we focus on *inclusion-free* instances which form a subset of the convex bipartite graphs. We will elaborate on the term inclusion-free in Section 4.2.1. Convex bipartite graphs (henceforward referred to as *convex graphs* for short) were introduced in [50] and are well known for their nice structures and both theoretical and practical properties. Many optimization problems become polynomial-time solvable or even linear-time solvable in convex graphs while remaining $\mathcal{NP}$-hard for general bipartite graphs [17]. Convex graphs can be recognized in linear time by using PQ-trees [17] and, moreover, the recognition algorithms provide the ordering $<_I$, given that the underlying graph is of course convex [15, 57, 87].

The interval case arises naturally in energy production applications where resources (energy) can be assigned and used within a number of successive time steps (that is the energy produced at some time step is available only for a limited period of time corresponding to an interval of time steps) and the goal is a fair allocation of the resources over time, *i.e.*, an allocation that maximizes the minimum accumulated resource we collect at each time step. In other words, we would like to have an allocation that guarantees the energy we collect at each time step is at least t, a pre-specified threshold. See also [100] for some applications in on-line scheduling.

4.1.1 Related Work

For the general Max-Min fair allocation problem, where a given item does not necessarily have the same value for each player (i.e., every player has her own value for any item), no "good" approximation algorithm is known. In [13], by using the natural LP formulation for the problem and similar ideas as in [83], an additive ratio of $\max_{i,j} v_{ij}$ is obtained, which can be arbitrarily bad. A stronger LP formulation, the *configuration LP*, is used to obtain a solution at least opt/n in [10]. Subsequently, [6] provided a rounding scheme for this LP to obtain a feasible solution of value no worse than $\mathcal{O}(\frac{\text{opt}}{\sqrt{n}(\log^3 n)})$. Recently in [98], an $\mathcal{O}(\sqrt{\frac{\log\log n}{n\log n}})$ approximation algorithm was proposed, which is close to the *integrality gap* of the configuration LP. On the negative side, there exists a simple $\frac{1}{2}$ inapproximability result [13] using the same ideas as in [83]. Due to the difficulty of the general case, subsequent research focused more on special cases. For the *restricted* case of the Max-Min allocation problem (also known as the restricted Santa Claus problem), where $v_{ij} \in \{0, v_j\}$ for $i \in [n]$ and $j \in [m]$, there is an $\mathcal{O}(\frac{\log\log\log n}{\log\log n})$ factor approximation algorithm [10]. Furthermore, the $\frac{1}{2}$ inapproximability result for the general case [13] carries over to this restricted case as well.

Recently, Feige proved that the integrality gap of the configuration LP defined and studied in [10], cannot be worse than some constant. In [5] an integrality gap of $\frac{1}{5}$ was shown for the same LP which was later improved to $\frac{1}{4}$. This implies that a $\frac{1}{4}$-approximation algorithm based on rounding the configuration LP for the restricted Max-Min allocation is possible, although no such algorithm is available yet.

The authors provide a local search heuristic with an approximation guarantee of $\frac{1}{4}$. However, this heuristic was not known to run in polynomial time. Later, it was shown in [94] that the local search can be done in $n^{\mathcal{O}(\log n)}$ time. In [58] the authors provided a constructive version of Feige's original nonconstructive argument based on Lovász Local Lemma, thus providing the first constant factor approximation for the restricted Max-Min fair allocation problem. They provide an α-approximation algorithm for some *constant* α where an explicit value of α is not provided (but is thought to be a huge constant). Thus there is still a gap between the $\frac{1}{2}$ inapproximability result and the constant α approximability result in [58]. Very recently, a 13-approximation was given for the problem [3], which provides the first constant factor polynomial time approximation algorithm for the problem for a particular constant value. Their approach uses, in a highly non-trivial way, the local search technique for hypergraph matching that was used in [5]. Another important aspect of this approach is that the algorithm is purely combinatorial.

Several special cases of the Max-Min fair allocation problem have been studied. The case where $v_{ij} \in \{0, 1, \infty\}$ is shown to be hard in [74] and a trade off between the running time and the approximation guarantee is established. In particular, for a number $\alpha \leqslant \frac{|I|}{2}$, it is shown how to design an $\alpha \cdot \frac{opt}{n}$-approximation algorithm in time $|P|^{\mathcal{O}(1)}|I|^{\mathcal{O}(\alpha)}$. In [11] the

authors consider the case in which each item has positive utility for a bounded number of players D, and prove that the problem is as hard as the general case for $D \leqslant 3$. They also provide a $\frac{1}{2}$ inapproximability result and a $\frac{1}{4}$ approximation algorithm for the *asymmetric* case (where each item has two distinct non-zero values for the two players interested in that item) when $D \leqslant 2$. The authors also provide a simpler LP formulation for the general problem and devise a polylogarithmic approximation algorithm that runs in quasipolynomial time (and a $|P|^{\varepsilon}$-approximation algorithm that runs in $|P|^{\mathcal{O}(\frac{1}{\varepsilon})}$ time, for some $\varepsilon \geqslant 0$). The same result has been obtained in [21], which includes a $\frac{1}{2}$ approximation when $D \leqslant 2$, thus matching the bound proved in [11]. In [107], the author provides a PTAS for a (very) special case of the problem considered in this chapter, namely, when the instance graph of the problem is a complete bipartite graph. In [86] a $\frac{1}{2}$-approximation algorithm was developed for the class of instances considered here, namely for the case where the intervals for each player are inclusion-free in the same sense introduced in this thesis. See also [88, 99] for some other special cases that our results generalize.

The Min-Max-Allocation problem, or the $R\,||C_{\max}$ problem, as it is known in standard scheduling notation, is an important class of resource allocation problems. We seek an allocation (also known as assignment) of jobs (the resources) to machines (the players) such that the makespan (the time by which the latest machine finishes its processing) is minimized.

For the $R\,||C_{\max}$ problem, a 2-approximation algorithm based on a characterization of the extreme point solutions of the natural linear programming relaxation of the problem is given [83]. The authors also provide a $\frac{3}{2}$ inapproximability result. These results have been adapted to the Max-Min case in [13]. So far, all efforts to improve either of the bounds have failed. In a very recent result [102], it is shown that the restricted version of $R\,||C_{\max}$ (where the processing time of a job j is v_j for a subset of the machines and infinity otherwise) admits an α approximation guarantee for α strictly less than 2. This result is an *estimation* result i.e., it estimates the (optimal) makespan of the restricted $R\,||C_{\max}$ within a factor of $\alpha = \frac{33}{17} + \varepsilon$ for some arbitrary small positive ε, although no polynomial time algorithm with such a performance guarantee is known. In [35], a 1.75 approximation algorithm is given for the restricted $R\,||C_{\max}$ problem where each job can be assigned to at most 2 machines. In this article, the authors claim that their 1.75 approximation for this restricted case is the first one that improves the factor 2 approximation of [83] on unbounded number of machines and jobs. Our PTAS thus provides a certain strengthening of their claim, providing the fist natural and non-trivial instance (as in the case of a complete bipartite graph) for which a PTAS is provided.

We note that further restrictions of this special case, where every job has degree at most two, have been studied [80]. Thus if the underlying bipartite graph is a tree, then a PTAS can be designed for the problem. If the processing times are in the set $\in \{1, 2\}$, then a $\frac{3}{2}$-approximation algorithm exists, which is the best possible unless $\mathcal{P} = \mathcal{NP}$. Finally, [105]

provides better approximations for several special cases of the problem. More importantly, it shows that the configuration LP for this restriction has an integrality gap of 2.

4.1.2 Our Contribution

We summarize our results below:

(1) We start by presenting a PTAS for the restricted Max-Min fair allocation problem when the instance of the problem is an inclusion-free convex graph, that is, the neighborhood of each player is an interval of items (Section 4.2). Notice that this instance of the problem is (strongly) $\mathcal{NP}$-hard, as it contains complete bipartite graphs as a special case (each player is adjacent to all the items), which is known to be strongly $\mathcal{NP}$-hard [46].

(2) Next, we modify our approach for the Max-Min allocation problem to obtain a PTAS for the $R\,|\,|\,C_{\max}$ problem with inclusion-free convex graphs. (Section 4.3).

To obtain the PTAS for the instances considered in this chapter, we first use scaling and rounding to classify the items into small and big items. Then, we prove that in a given instance of the problem, for any assignment of a certain value, another assignment of slightly less value but with simpler structure exists. Finally, we provide an algorithm that searches all these simple-structured assignments.

4.2 Max-Min Allocation Problem (Santa Claus) on Convex Graphs

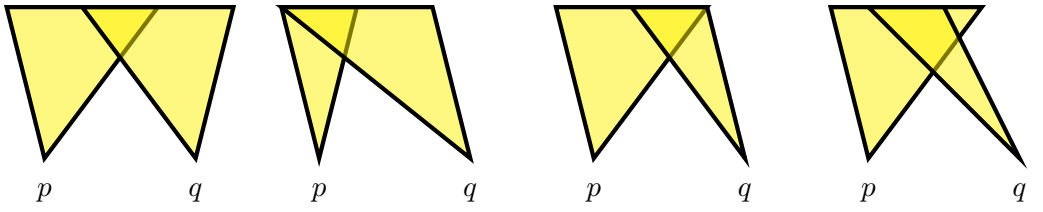
As instance of the problem is given via a convex graph $H = (I,\, P,\, E)$ and a utility function $val : I \to \mathbb{Q}_{>0}$ which associates a value v_i to every item x_i in I. Structural properties and algorithms of convex graphs have received attention in the field of algorithmic graph theory. We refer the reader to texts such as [17] and [101]. For the sake of completeness, we give a definition of this class of graphs and introduce some notations that we will use throughout the chapter.

4.2.1 Preliminaries and Notations

We first introduce the notation that we will use throughout this chapter.

Notation of the Chapter

Throughout the chapter, we use **bold** mathematical symbols to refer to vectors. Superscripts are reserved for player indices. For simplicity, we denote the induced subgraphs of the input convex graph H by $H' = (I',\, P')$ for some $I' \subseteq I$ and $P' \subseteq P$ and refrain from explicitly



(a) Overlapping **(b)** Left Inclusive **(c)** Right Inclusive **(d)** Margined

Figure 4.1: Four types of intersecting intervals of two players

mentioning the subset of edges $E' \subseteq E$. The set E' contains all those edges of E that are incident to at least one vertex in I' and at least one vertex in P'. The underline is used for emphasis. Finally, we use *italics* to highlight the terms we define in the chapter.

Preliminary Definitions

To define the class of convex graphs we first need to introduce the following property.

Definition 13 (Adjacency Property)**.** *[17] Let $H = (X, Y, E)$ be a bipartite graph. An ordering $<_X$ of X in H has the* adjacency property *if for each vertex $y \in Y$, the neighbourhood of y in X denoted by $N(y)$ consists of vertices that are consecutive in $<_X$.*

A convex graph is defined as follows.

Definition 14 (Convex Graph)**.** *[17] A bipartite graph $H = (X, Y, E)$ is convex if there exists an ordering of X or Y that satisfies the adjacency property.*

In this chapter, we deal with a subclass of convex graphs known as inclusion-free convex graphs. In this subclass, inclusion may still occur between the intervals, but it should follow certain rules. We first explain the rules with the help of Figure 4.1. Assume that a convex graph $H = (I, P, E)$ is given as the input instance. Consider two players p and q in P. Their neighbourhoods in H can either be separate (their intersection is empty), or intersecting. If the neighbourhood of two players p and q is intersecting, four types of intersection between the two neighbourhoods may occur. These types are depicted in Figure 4.1. We say that two intervals are *properly overlapping* (Figure 4.1 **(a)**) if neither of the two intervals contains the other one, or *left inclusive* (Figure 4.1 **(b)**) if one interval contains the other and the two intervals share their left boundary, or 3) *right inclusive* (Figure 4.1 **(c)** if one interval contains the other and the two intervals share their right boundary, or 4) *margined-inclusive* (Figure 4.1 **(d)**) if one interval is completely contained in the other, and the intervals do not share their boundaries. The subclass of inclusion-free convex graphs forbids margined-inclusion while left inclusion and right inclusion may still occur.

We define the class of inclusion-free convex graphs as follows. Given an ordering of the items $<_I$ that satisfies the adjacency property, for any pair of players if the neighbourhood of

one player is completely included in the neighbourhood of the other, the smaller neighbourhood must either contain the leftmost or the rightmost item of the bigger neighbourhood with respect to the ordering $<_I$. This property is sometimes referred to as the *enclosure* property in the literature.

Definition 15 (Enclosure Property)**.** *Let $H = (X, Y, E)$ be a bipartite graph. An ordering $<_X$ of X in H has the enclosure property if for each pair of vertices $y, y' \in Y$ for which $N(y) \subseteq N(y')$ in X, the vertices of $N(y') \setminus N(y)$ appear consecutively in $<_X$. We say that the graph H as the enclosure property if such an ordering of the vertices of H exists.*

As we mentioned earlier, every convex graph admits such an ordering and we can find it in linear time. Let $<_I$ be the ordering of items in set I that satisfies the adjacency property. For every vertex $p \in P$ let $[\ell_p, r_p]$ be the interval of the items adjacent to p. Based on the ordering $<_I$ on I, we define the following ordering on P.

Definition 16 (Lexicographical Ordering of Players)**.** *For a given ordering $<_I$ of items I, an ordering of players is called lexicographical if and only if a player p is ordered before another player q whenever $\ell_p <_I \ell_q$, or $\ell_p = \ell_q$ and $r_p \leqslant_I r_q$ (breaking ties arbitrarily), in which $r_p \leqslant_I r_q$ means either $r_p = r_q$ or $r_p <_I r_q$.*

The adjacency property is equivalent to the the *Min ordering* property in bipartite graphs and the combination of both adjacency and enclosure properties is equivalent to the *Min-Max ordering* property. Here, we define these two properties.

Definition 17 (Min Ordering Property)**.** *A bipartite graph $H = (X, Y, E)$ fulfills the* Min ordering *property if there exist orderings of X and Y, $<_X$ and $<_Y$ respectively, which satisfy the following: if for the vertices $x <_X x'$ in X and $y <_Y y'$ in Y, we have that $xy' \in E$ and $x'y \in E$, then xy must also be an edge in E.*

Definition 18 (Min-Max Ordering Property)**.** *A bipartite graph $H = (X, Y, E)$ fulfills the* Min-Max ordering *property if there exist orderings of X and Y, $<_X$ and $<_Y$ respectively, which satisfy the following: if for the vertices $x <_X x'$ in X and $y <_Y y'$ in Y, we have that $xy' \in E$ and $x'y \in E$, then xy and $x'y'$ must also be edges in E.*

These properties are depicted in Figure 4.2. The two equivalencies can be derived from the previously known results. For instance, one can show that the adjacency and the enclosure property together results in Min-Max ordering from Theorem 5 in [62] and the definition Min-Max ordering in [55]. Note that not all convex graphs satisfy the enclosure (or Min-Max ordering) property. In fact, the class of convex graphs that satisfy the enclosure property is a proper subclass of the convex class and is known to be equivalent to two famous graph classes, the *bipartite permutation graphs* and the *proper interval bigraphs*. Therefore, we say that Bipartite Permutation $\subset$ Convex.

The following observation shows an interesting property of graphs that satisfy both the adjacency and enclosure properties.

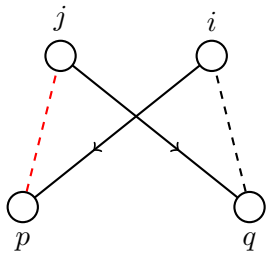


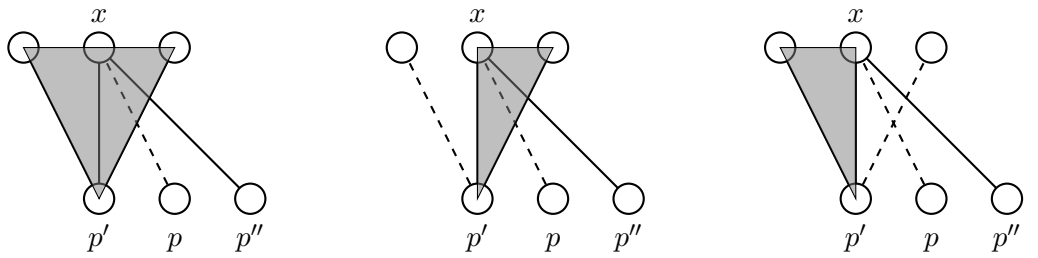
Figure 4.2: Assume that $p <_I q$ and $j <_I i$ in the ordering $<_I$ that satisfies the adjacency property. If the graph only has the left dotted connection between j and p, then it is said to have the min ordering property for players p and q. If the graph has both dotted lines, then it is said to have min-max ordering property over players p and q.

Observation 1. *Assume we are given a bipartite permutation (inclusion-free convex) graph $H = (I, P, E)$ alongside an ordering of items $<_I$ that satisfies both the adjacency and the enclosure properties. If I is ordered according to $<_I$, then the lexicographical ordering of P based on $<_I$ also satisfies both properties. In other words, when P is ordered lexicographically based on $<_I$, (i) the neighbourhood of every item $x \in I$ forms an interval in P, and furthermore, (ii) for every pair of items $x_1, x_2 \in I$, their respective neighbourhoods are either properly overlapping, left inclusive, or right inclusive.*

Proof. We first show part *(i)*. For the sake of contradiction, assume there is an item $x \in I$ such that the neighbourhood of x in P does not form an interval. This implies that a gap exists in the neighbourhood of x in P, meaning that there are players $p' <_P p <_P p''$ such that x in adjacent to p' and p'' but not p. Here, $<_P$ denotes the lexicographical ordering of payers. Since the neighbourhood of players form intervals in I, the neighbourhood of p' can include x in one of the three ways depicted in Figure 4.3. The dotted edges represented the non-edges. We go through each case:

Case 1: x is in the middle of the interval of p'. As p is not adjacent to x, we first consider the case in which $\ell_p <_I x$. We can conclude $r_p <_I x$ since the neighbourhood of p is an interval in I. This means that either $\ell_p = \ell_{p'}$ which together with the fact that $r_p <_I r'_p$ (note that p' is adjacent to x and p is not, so the right boundary of the neighbourhood of p' must be to the right of that of p in this case) means that p should come before p' in the lexicographical ordering contradicting the assumption that $p' <_P p$, or we have that $\ell_{p'} <_I \ell p$ and $r_p <_I r'_p$ which is contradicting the assumption that the graph satisfies the enclosure property (the neighbourhood of p falls completely inside that of p'). Therefore it follows that $x <_I \ell_p$. In this case, since $p <_P p''$, we either have that $\ell_p = \ell_{p''}$ or $\ell_p <_I \ell_{p''}$. In both this situations, p'' cannot be adjacent to x, implying that the assumption of Case 1 over the neighbourhood of players is false.

Case 2: x is on the left boundary of the interval of p'. In this case, we certainly have that $x <_I \ell_p$ as p is lexicographically larger than p' and not adjacent to x. As in Case 1, either



(a) x in the Middle **(b)** x on the Left Boundary **(c)** x on the Right Boundary

Figure 4.3: Different cases a gap may exist in the interval of an item x in an inclusion-free convex graph

$\ell_p = \ell_{p''}$ or $\ell_p <_I \ell_{p''}$ due to the lexicographical ordering. In both situations, p'' cannot be adjacent to x which makes Case 2 a contradiction as well.

Case 3: x is on the right boundary of the interval of p'. Again, $x <_I \ell_p$, as assuming otherwise leads to a contradiction we mentioned in Case 1. Therefore, since $p <_P p''$, we get the two cases as before, $\ell_p = \ell_{p''}$ or $\ell_p <_I \ell_{p''}$, both leading to a contradiction.

The assumption that a gap exists in the neighbourhood of x leads to contradiction in either case. This proves part *(i)* of the observation. For part *(ii)*, we assume there is an item x whose neighbourhood is marginally included inside the neighbourhood of some other item x'. Again, there are two cases: one in which $x' <_I x$, and the other $x <_I x'$. We only show the former as the latter follows from symmetry. Since the interval of x is entirely inside the neighbourhood of x' without touching any of the boundaries, there must exist players $p' <_P p <_P p''$ such that p is adjacent to both x and x', but p' and p'' are only adjacent to x', lying in the left and right side of the neighbourhood of x. Figure 4.4 depicts the neighbourhood of players and items, with the dashed edges representing the non-edges. Note that the neighbourhood of each of the players is an interval in I. Players p' and p'' are both adjacent to x' but not x, while player p is adjacent to x. Since $x' <_I x$, we conclude that $\ell'_p <_I r'_p <_I r_p$ and $\ell''_p <_I r''_p <_I r_p$. This contradicts with the assumption that p'' is lexicographically greater than p, therefore the intervals of items in the set P must be inclusion-free. □

Remark 3. *Note that for any bipartite permutation graph H, we know that H is* bi-convex*, i.e., there is an ordering of X and Y that satisfies the adjacency property, or* Bipartite Permutation ⊂ Biconvex ⊂ Convex*. Observation 1 suggest a yet stronger property; there is an ordering of X and Y that fulfills both the adjacency and enclosure properties.*

Given a convex graph $H = (I, P, E)$, and ordering $<_I$ of items that satisfies the adjacency property, and a target value t, our goal is to find a t-assignment of items to players, defined informally as follows:

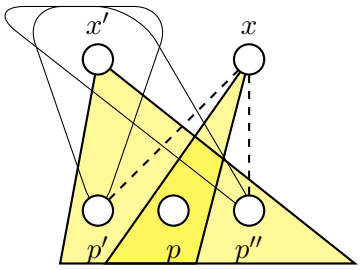


Figure 4.4: The contradictory assumption that intervals of items in the set of players P are not inclusion-free

Definition 19 (t-assignment for the Max-Min case)**.** *A t-assignment, for any $t \geqslant 0$, is a feasible assignment such that every player p_j receives a set of items $I^j \subseteq [\ell_{p_j}, r_{p_j}]$ with total value at least t.*

In the Min-Max case, a t-assignment is simply a feasible assignment such that each player (machine) receives total utility at most t.

In this chapter, we only deal with the easier case of inclusion-free instances. All the definitions and theorems in later sections apply to the class of inclusion-free convex graphs (unless mentioned otherwise explicitly). To the best of our knowledge, no PTAS is known for general instances with margined-inclusive intervals, where the convex graph only satisfies the adjacency property. Whether there exists a PTAS for convex graphs in general is an interesting research question that we pose as an open problem.

4.2.2 Preprocessing the Instance

Given a particular instance of the problem, we first simplify the input instance. For a positive rational number t, we may assume that the value of each item is at most t. If item x_i has value $v_i > t$ then we set v_i to t without loss of generality. By a proper scaling, *i.e.*, dividing each value by t, we may assume that the value of each item is in $[0, 1]$. Observe that a t-assignment becomes a 1-assignment. We do a binary search to find the largest value of t for which a t-assignment exists (in the Min-Max case we seek the smallest t such that a t assignment exists). The binary search is carried out in the interval $[0, \frac{1}{n}\sum_{i=1}^{m} v_i]$ where 0 is an absolute lower bound, and $\frac{1}{n}\sum_{i=1}^{m} v_i$ is an absolute upper bound of the optimal solution respectively (for the Min-Max case the binary search is carried out in the interval $[\frac{1}{n}\sum_{i=1}^{m} v_i, \sum_{i=1}^{m} v_i]$).

For a subset $P' \subseteq P$ of players, let $N_{[H]}(P')$ be the union of the set of all neighbours of the players in P' in the graph H. We let $N(P')$ denote this set whenever the graph H is implied by the context. For an interval $[i, j]$ of items, let $P[i, j]$ be the set of players whose entire neighbourhood lies in $[i, j]$, that is $P[i, j] = \{p \in P : N(p) = [\ell_p, r_p] \subseteq [i, j]\}$. For a subset $I' \subseteq I$ of items, let $val(I')$ denote the sum of values of all the items in I'. We note that in every 1-assignment, for every subset $P' \subseteq P$ of players, the sum of the value of items

in its neighbourhood should be at least $|P'|$. In other words, $\forall P' \subseteq P : val(N(P')) \geqslant |P'|$. If the value of each item is 1, then this condition is the well known Hall's condition [59], a condition sufficient and necessary for a bipartite graph to have a perfect matching. We consider a more general version of the 1-assignment and derive a generalized version of Hall's condition below. Each player $p \in P$ has a demand $d(p)$. This version contains the 1-assignment as a special case, i.e., the case where $d(p) = 1$ for all players $p \in P$. Also, for a subset of players $P' \subseteq P$, let $d(P')$ denote the sum of demands of all players in P'. Now the generalized Hall's condition (for the Max-Min case) becomes: $\forall P' \subseteq P : val(N(P')) \geqslant d(P')$. From now on we refer to this condition as Hall's condition. Note that this condition is necessary to satisfy the players' demands, but not sufficient (see Figure 4.5). Lemma 3 shows that in order to check Hall's condition for H it suffices to check it for every interval of items. Therefore, Hall's condition in our setting becomes Condition 4.2.1 below:

$$\forall\ [\ell, r] \subseteq [1,\ m] : \qquad val([\ell, r]) \geqslant d(P[\ell, r]). \tag{4.2.1}$$

Lemma 3. *In order to check Hall's condition for H it suffices to verify Condition 4.2.1. In other words, it suffices to check Hall's condition for every set of players $P[\ell, r]$, $[\ell, r] \subseteq [1,\ m]$.*

Proof. Assume we are given a convex graph H. We claim that it is sufficient to check Hall's condition only for intervals of items, meaning that if there is any violations of Hall's condition, then there is at least one violation over an interval of items. In other words, we show that there for a player P' for which *(i)* $N(P')$ consists of several maximal intervals (therefore $N(P')$ is not an interval itself), if *(ii)* Hall's condition is violated for P', then there exists an interval of items $[\ell,\ r] \subseteq I$ for which Condition 4.2.1 is violated.
Assume that there exist several maximal intervals $J_1,\ J_2,\ \ldots,\ J_t$ whose union gives $N(P')$. Since each player is adjacent to an interval of items, there exists a corresponding partition of P' into subsets $P'_1,\ P'_2,\ \ldots,\ P'_t$, such that $N(P'_i) = J_i$. Since Hall's condition is violated, we have

$$val(J_1) + \ \ldots\ + val(J_t) = val(N(P')) < d(P') = d(P'_1) + \ \ldots\ + d(P'_t)$$

Thus, there must exist an $i, 1 \leqslant i \leqslant t$, for which $val(J_i) = val(N(P'_i)) < d(P'_i)$. Now let ℓ and r be the leftmost and rightmost items in J_i respectively. Since $N(P'_i) = J_i = [\ell,\ r]$, then the neighbourhood of every player in P'_i should fall entirely in the interval $[\ell,\ r]$. Therefore, $P'_i \subseteq P([\ell,\ r])$. Thus, we conclude $val([\ell,\ r]) < d(P'_i) \leqslant d(P([\ell,\ r]))$, which is a violation of Condition 4.2.1 for the interval $[\ell,\ r]$. □

As a result of Lemma 3, it is sufficient to check Hall's condition for every interval of items. Since there are at most m^2 intervals, Hall's condition can be verified in time polynomial in size of H.

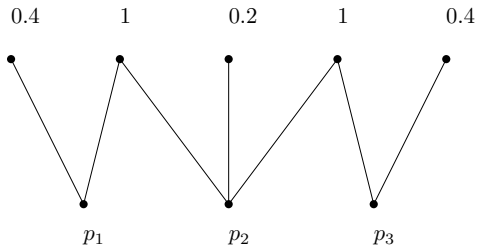


Figure 4.5: An instance in which Hall's condition is satisfied for $t = 1$ but the optimal solution value is not greater than 0.4. In this examples, $d(p_1) = d(p_2) = d(p_3) = 1$.

Rounding the Instance

In a given instance H of the problem, we round the item values to obtain a rounded instance H'. For any integer $k \geqslant 4$, we let $\frac{1}{k}$ be the error parameter. For each instance for which there is an optimal 1-assignment, we seek an assignment such that each player receives a set of items with total value at least $1 - \frac{4}{k+1}$, $k \geqslant 4$.

- We call an item x_i *small* if its value is less than or equal to $\frac{1}{k}$. The values of small items are not changed in H'.
- We call an item *big* if its value is greater than $\frac{1}{k}$. If the value of item x_i, v_i, is in the interval $(\frac{1}{k}(1+\frac{1}{k})^{\tau}, \frac{1}{k}(1+\frac{1}{k})^{\tau+1}]$ for $\tau \geqslant 0$, it is rounded to $\frac{1}{k}(1+\frac{1}{k})^{\tau+1}$ in H'.

After the rounding, there are at most $C = \lceil \log k/(\log(1+\frac{1}{k})) \rceil$ *categories* of big items, that is items with distinct values more than $\frac{1}{k}$. One can easily verify that C is no more than $k^{1.4}$. For $0 \leqslant \tau \leqslant C$, let $q_\tau = \frac{1}{k}(1+\frac{1}{k})^{\tau}$. For subset I' of I let $v_s(I')$ denote the value of the small items in I'. The rounding process in the Min-Max case is slightly different and we include it in the corresponding section.

4.2.3 The Algorithm for Inclusion-Free Convex Graphs

In this section, we consider the cases in which the neighbourhoods of no two players form a pair of margined- inclusive intervals. Note that left and right inclusion may still occur in such instances. The main result of this section is the following theorem.

Theorem 1. *Let H be an instance of the problem before rounding with n players and m items. Then for $k \geqslant 4$ there exists a $(1 - \frac{4}{k+1})$-approximation algorithm for the Max-Min allocation problem on inclusion-free convex graphs with running time $\mathcal{O}\left((m+n)nm^{2(C+1)}\right)$, in which $C \leqslant k^{1.4}$.*

Our proof technique builds upon a previous work of Alon et al. [2]. In particular, we use the notion of "input vectors" (Definition 20) in a dynamic programming algorithm that considers the players one by one and remembers plausible allocations for them that can lead to a $(1-\varepsilon)$-assignment. Input vectors are configuration vectors that indicate the number

of available items in an instance of the problem or one of its subproblems. In their paper, Alon et al. study complete bipartite graphs, *i.e.*, the case where all items are available to all players. This fact allows them to ignore the actual assignment of items to players in their dynamic programming algorithm and only work with the configuration vectors for the set of available items, *i.e.*, the input vectors; once the configuration vector for an assignment of items to a certain players is known, any set of items that matches the configuration vector can be assigned to the player. In other words, all items of a certain size are identical to the algorithm. Since the number of all possible input vectors is polynomial in the instance size, they manage to provide a polynomial-time algorithm in the end.

Our work focuses on a more general case – that of inclusion-free convex graphs which contains complete bipartite graphs as a subset. In this setting, an item may be adjacent only to a subset of players. As a result, to identify a subproblem, we cannot simply represent the set of available items to each player with the corresponding input vector. On the other hand, one cannot consider all possible ways of assigning items to players since the number of such assignments is exponential and computationally prohibitive. For the case of margined inclusion- free instances, we prove one can still identify the subproblems from their respective input vectors. To that end, we introduce $(1-\varepsilon)$-assignments of a certain structure (which we call simple-structured assignments) and prove their existence in any instance of the problem which admits a 1-assignment. Such assignments are defined in Definitions 21 and 25, and shown to exist in Lemmas 4 and 5.

In this section, we assume that the convex graph H is a rounded input. For a rounded instance, it suffices to set the constant parameter ε to $\frac{3}{k}$. In the proof of Theorem 1, we show how this bound guarantees a $1-\frac{4}{k+1}$ approximation factor for the original instance. The dynamic programming algorithm has a forward phase and a backward phase. In the forward phase, the algorithm checks the existence of a solution by filling up a table. During this phase, the algorithm runs n *steps*, one for each player. At a given step j, it checks for feasible $(1-\varepsilon)$-assignments $\boldsymbol{\mathcal{A}}' = (I^{n-j+1}, I^{n-j+2}, \ldots, I^n)$ for players $p_{n-j+1}, p_{n-j+2}, \ldots, p_n$, called *partial* assignments, based on the solutions of the sub-problems in the previous step (feasible $(1-\varepsilon)$-assignments for players $p_{n-j+2}, \ldots, p_n$). During the first step ($j = 1$), the algorithm searches for $(1-\varepsilon)$-assignments for player p_n from scratch, therefore this case is treated differently. The backward phase also has n steps. During this phase, the algorithm considers the players in a reverse order of the forward phase, *i.e.*, from p_1 to p_n, and generates the final set of items to be allocated to each player based on the information stored in the table.

Every partial assignment of the forward phase indicates a potential assignment of items to players[1]. After every such assignment, the set of available items changes, thus a subproblem is instantiated. The induced subgraph of H on the set of available items and unsatisfied players is called a *remainder* graph. The algorithm regards every remainder graph as an instance of a subproblem of the original allocation problem. Note that the number of possible

[1]The assignment will be finalized once the algorithm finishes the steps of the forward phase.

partial $(1-\varepsilon)$-assignments and their corresponding remainder graphs can be exponential. Some remainder graphs may eventually result in a solution and some may not. Thus, at each step, a naïve dynamic programming approach might need to remember the remainder graphs, which will lead to an exponential memory (and time) complexity. By exploiting the properties of convex graphs (specifically the Min-Max ordering) we can show that one can do better than the naïve approach. In this section, we examine some structural properties of the problem that allow us to find a solution for any given instance while checking a polynomial[2] number of simple-structured assignments. Our dynamic programming algorithm uses input vectors, which we introduce here.

Definition 20 (Input Vector)**.** *For a given convex graph H for an input instance with ν_τ big items of size q_τ for $\tau = 1, 2, \ldots, C$ and small items of total value in the interval $(\frac{\nu_0-1}{k}, \frac{\nu_0}{k}]$, an input vector is a configuration vector of the form $\nu(H) = \boldsymbol{\nu} = (\nu_0, \nu_1, \ldots, \nu_C)$.*

Note that for an arbitrary input vector $\boldsymbol{\nu} = (\nu_0, \nu_1, \ldots, \nu_C)$, $\nu_\tau \leqslant m$ for all big items of size q_τ, $1 \leqslant \tau \leqslant C$ as there can be at most m items of a certain value. Furthermore, ν_0 is the total value of the small items in integral multiples of $\frac{1}{k}$ which has been rounded up. Therefore, $\nu_0 \leqslant m$ since $\frac{m}{k}$ is an upper bound on the total value of the small items. As a result, there are at most m^{C+1} possible input vector values, which is a polynomial in the size of the problem instance. In the rest of this chapter, we let $\mathcal{V}$ denote the set of all possible input vectors. Similar to the algorithm by Alon et al., we are interested in a dynamic programming approach that only deals with the input vectors in $\mathcal{V}$ rather than the (potentially exponential) remainder graphs. Unlike the case of complete graphs, we cannot assign items of a certain size interchangeably. However, by imposing the right set of restrictions on the assignments we are able to retrieve (reconstruct) a remainder graph from any input vector whose total sum of items is almost equal to the sum of values indicated by the input vector. In particular, we wish to show the following two facts:

Fact 4.2.1. *Whenever there exists an arbitrary 1-assignment of items to players for an instance of the problem, there also exists a restricted $(1-\varepsilon)$-assignment.*

Fact 4.2.2. *Given that a 1-assignment exists for an instance of the problem, then there also exists a polynomial time algorithm that, given the input vectors for every step (partial assignment) of a $(1-\varepsilon)$-assignment, reconstructs the remainder graph of every step in such a way that the total value of the items in each reconstructed remainder graph is*

- *exactly the same as that of the corresponding remainder graph in the original (and unknown) 1-assignment for every category $\tau = 1, 2, \ldots, C$ of the big items and,*
- *only a small fraction ε more than its counterpart in the original remainder graph for the small items.*

[2]In the number of players and items, but exponential in the inverse of the error parameter.

As the first of the restrictions we impose on the assignment to obtain remainder graphs of simpler structures, we define *right-aligned* assignments.

Definition 21 (Right-Aligned Assignment)**.** *For a given convex graph H, an ordering of items $<_I$ that satisfies the adjacency property, and a lexicographical ordering of players with respect to $<_I$, an assignment of items to players $\boldsymbol{\mathcal{A}} = (I^1, I^2, \ldots, I^n)$ is called right-aligned if and only if it satisfies the following properties recursively:*

(1) Let p_n be the last (rightmost) player in the graph. All the big items of value $q_\tau = \frac{1}{k}(1+\frac{1}{k})^\tau$ $(1 \leqslant \tau \leqslant C)$ assigned to p_n are the rightmost ones with that value in $N_{[H]}(p_n)$. Furthermore, all the small items assigned to p_n are also the rightmost ones in her neighbourhood.

(2) In the remainder graph $H' = (I \setminus I^n, P \setminus p_n)$, that is if p_n and all the items assigned to her are removed from the graph H, the rest of the assignment, $\boldsymbol{\mathcal{A}'} = (I^1, I^2, \ldots, I^{n-1})$, is a right-aligned assignment.

(3) An empty assignment (i.e., when no players are left in the graph) is considered a right-aligned assignment.

Next, we prove Fact 4.2.1 for the right-aligned assignments in a lemma called the *Alignment Lemma* (Lemma 4). Intuitively, the lemma states that if a 1-assignment exists, then there exists a $(1-\varepsilon)$-assignment with items aligned to the right, in which $\varepsilon = \frac{1}{k}$. We first need the following observation and definition.

Observation 2. *Convex graphs have the hereditary property, meaning that every induced subgraph of a convex graph is also convex.*

This is straightforward to see. Assume that graph $H = (X, Y, E)$ is convex. Without loss of generality assume X has an ordering that respects the adjacency property, meaning the neighbourhoods of members of X in Y are intervals. Regardless of how we remove nodes from these intervals in any subgraph of H, the remaining parts of the neighbourhoods still form intervals.

Definition 22 (Assignment Vector)**.** *For an assignment of items to the set of players $\boldsymbol{\mathcal{A}} = (I^1, I^2, \ldots, I^n)$, and for a player p_j, $1 \leqslant j < n$, let $\boldsymbol{\alpha}^j = \nu(H^j)$ for $j = 1, 2, \ldots, n$, in which $H^j = (P \setminus \{p_n, p_{n-1}, \ldots, p_{j+1}\}, I \setminus (I^n \cup I^{n-1} \cup \ldots \cup I^{j+1}))$ is the remainder graph for player p_j in the assignment $\boldsymbol{\mathcal{A}}$ (for $j = n$, let $H^n = H$). We call the vector $\boldsymbol{\alpha}^j = (\alpha^j_0, \alpha^j_1, \ldots, \alpha^j_C)$ the* assignment vector *of player p_j in the assignment $\boldsymbol{\mathcal{A}}$. Furthermore, let $\boldsymbol{\alpha} = (\boldsymbol{\alpha}^1, \boldsymbol{\alpha}^2, \ldots, \boldsymbol{\alpha}^n)$ be the assignment vector for the entire assignment $\boldsymbol{\mathcal{A}}$.*

Lemma 4 (The Alignment Lemma)**.** *Suppose there exists a 1-assignment for a given convex graph H. Then, there also exists a <u>minimal</u> $(1-\frac{1}{k})$-right-aligned assignment for H. Furthermore, the two assignments will have identical assignment vector.*

Proof. We let $\boldsymbol{\mathcal{A}} = (I^1, \ldots, I^n)$ denote the 1-assignment stated in the lemma and let $\boldsymbol{\alpha}$ be it assignment vector. Also, we denote by I^j_τ the set of big items of value q_τ assigned to player p_j for $1 \leqslant \tau \leqslant C$ and by I^j_0, the set of small items allocated to p_j in the assignment $\boldsymbol{\mathcal{A}}$. We prove the lemma by transforming the assignment $\boldsymbol{\mathcal{A}}$ into a minimal $(1 - \frac{1}{k})$-right-aligned assignment, $\boldsymbol{\mathcal{A}}' = (J^1, J^2, \ldots, J^n)$. $\boldsymbol{\mathcal{A}}'$ is minimal in the sense that the removal of any items from the sets $J^1, \ldots, J^n$ will cause its value to drop below $1 - \frac{1}{k}$. This task is carried out in two independent and consecutive rounds. In round 1, the big items assigned to each player are aligned to the right while the small items are aligned in round 2. Each round consists of n steps, one for each player. The two rounds are independent of each other in that the alignment of the big items in the first round has no impact on the alignment of small items in the second round. In the following, we assume the players are ordered lexicographically based on the ordering of items $<_I$.

Round 1: In round 1, the big items are right-aligned in n steps, where in each step, the big items of a single player are aligned in C *micro-steps*, one micro-step for each item category. We initialize j to be n, indicating that we start with the last player, p_n, and let τ be 1. Also, let H^j denote the remainder graph at the beginning of step j. At the τ^{th} micro-step, we only look at items of size q_τ in the graph and replace the set I^j_τ with its right-aligned counterpart J^j_τ. Note that according to Observation 2, the induced subgraph on items of a certain size, say H^j_τ, forms a convex graph. We start with $J^j_\tau = I^j_\tau$. If I^j_τ is right-aligned, *i.e.*, all items of value q_τ assigned to p_j are the rightmost ones in $N_{H^j_\tau}(p_j)$, then we return J^j_τ. If however, I^j_τ is not right-aligned, there must be two items x_r and x_t of the same value q_τ, where $x_r <_I x_t$ and x_r is assigned to p_j, but x_t is not. If x_t is not assigned to any other player, then we can simply assign it to p_j instead of x_r. If it is assigned to some player p_ℓ, x_r must also be connected to p_ℓ. Otherwise p_ℓ would come after p_j in the lexicographical ordering of the players. Since the two items have the same value, we use the adjacency property to swap them in the assignment. Thus, p_j gets x_t and p_ℓ gets x_r in the transformed assignment J^j_τ. Now we have one less item out of the alignment. We continue this process until there are no items in J^j_τ out of the alignment. Then we proceed to the next micro-step by setting $\tau \leftarrow \tau + 1$. At the end of C micro-steps, we let J^j be $\bigcup_{\tau=1}^{C} J^j_\tau$. Furthermore, we obtain the remainder graph $H^{j-1} = (I \setminus J^j, P \setminus \{p_j\})$. This graph will be the input to the next step of the alignment. We now proceed to the next step by updating $j \leftarrow j - 1$. Note that the assignment vector for $\boldsymbol{\mathcal{A}}' = (J^1, J^2, \ldots, J^n)$, say $\boldsymbol{\alpha}'$, is identical to $\boldsymbol{\alpha}$ on the big item coordinates since for any arbitrary player p, we did not change the number of big items of any category assigned to p. As a result, we have not lost any solution quality.

Round 2: In round 2, we obtain a right-aligned assignment for small items. We first explain the procedure, and then prove its correctness. Let H_0 denote the subgraph of H that contains only small items. The neighbourhood of each player in H_0 is still an interval (by Observation 2). For $j \in [n]$, let $1 - w_j$ be the total value of the big items assigned to

player p_j in the 1-assignment. Therefore, w_j denotes the deficit of player p_j that should be satisfied by small items. If the demands of each player p_j is set to w_j for $j \in [n]$, then H_0 must satisfy Hall's condition with these demands. This is due to the fact that a 1-assignment exists in the original graph for which Hall's condition is necessary, and that during the first round, the values of the big items assigned to the players did not change. Following the idea of round 1, we align the items in n steps. We start with the last player, so we let j be n and let I_0^j denote the set of small items assigned to p_j. We also let H_0^j denote the remainder graph at the beginning of step j. Note that $val(I_0^j) = w_j$. We replace this set with the set of right-aligned items, J_0^j, of value at least $w_j - \frac{1}{k}$. We first let J_0^j be the empty set. Then, we pick small items from the neighbourhood of p_j in H_0^j and add them to J_0^j. We continue to do this as long as $val(J_0^j) \leqslant w_j - \frac{1}{k}$. When the algorithm stops, since the value of each small item is at most $\frac{1}{k}$, $w_j - \frac{1}{k} < val(J_0^j) \leqslant w_j$. By doing this, we ensure that player p_j gets a value strictly greater than $w_j - \frac{1}{k}$ in small items, and strictly greater than $1 - \frac{1}{k}$ in total. The remainder graph for the next step of round 2, H_0^{j-1}, is obtained by $H_0^{j-1} = (I \setminus J_0^j, P \setminus \{p_j\})$. Note that each such remainder graph is still a convex graph because we started with a convex graph H_0 and at each step, we removed items from right to left in the ordering $<_I$. Also note that the small coordinate of the assignment vectors of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{A}}'$ ($\boldsymbol{\alpha}$ and $\boldsymbol{\alpha}'$ respectively) are identical too, which means $\boldsymbol{\alpha} = \boldsymbol{\alpha}'$. The reason is that we packed the small items in $\boldsymbol{\mathcal{A}}'$ is such a way that they have the same total value as in $\boldsymbol{\mathcal{A}}$ when counted in integral multiples of $\frac{1}{k}$ and rounded up. We then move to the next step by updating $j \leftarrow j - 1$.

To prove the correctness, we show that after each step j, $1 \leqslant j \leqslant n$, we can obtain a right-aligned assignment of small items the sum of whose values is strictly greater than $w_j - \frac{1}{k}$ for player p_j. Recall that we chose every J_0^j ($1 \leqslant j \leqslant n$) to be the minimal right-aligned set of items assigned to player p_j for which $w_j - \frac{1}{k} < val(J_0^j) \leqslant w_j$. For the sake of contradiction, assume that for some j, $1 \leqslant j \leqslant n$, we cannot provide a right-aligned assignment of small items J_0^j for which $val(J_0^j) > w_j - \frac{1}{k}$. Let t be the largest such index. This assumption implies that $val(N_{[H_0^t]}(p_t)) \leqslant w_t - \frac{1}{k}$. We let $H_0^{t,n}$ denote the induced subgraph of H_0 on players p_t, p_{t+1}, …, p_n and the small items in their neighbourhoods. We consider the partial right-aligned assignments of small items in $H_0^{t,n}$ represented by J_0^t, J_0^{t+1}, …, J_0^n, and claim that for a subset of the players p_t, p_{t+1}, … p_n, Hall's condition is violated for small items in the original graph H_0. Since Hall's condition is necessary for any feasible assignment that fulfills the demands of players, this in turn implies that there is no 1-assignment for the instance, which contradicts our earlier assumption. To prove this claim, we introduce the notion of a *gap* in the assignment. With respect to an ordering of items $<_I$, a gap exists in a (partial) assignment if x_j is not assigned to any player, but there exists another item x_i, such that $x_i < x_j$ in $<_I$ and x_i is assigned to some player. The item x_j is said to be in the gap with respect to the partial assignment. Based on this notion, we consider two cases:

Case 1: There exists a gap in the partial assignment. Let u be the smallest index in $N_{[H_0^{t,n}]}(p_t, p_{t+1}, \ldots, p_n)$ for which item x_u is in the gap for the partial right-aligned assignment. By the choice of u, x_{u-1} must be assigned to a player in the set $\{p_t, p_{t+1}, \ldots, p_n\}$. Let p_v be that player. Note that $x_u \notin N_{[H_0^{t,n}]}(p_v)$ as otherwise x_u would be assigned to p_v in the right-aligned assignment. Item x_{u-1} is thus the rightmost item assigned to player p_v. Let x_ℓ and x_r be the left-most and the right-most items in $N_{[H_0^{t,n}]}(p_t, p_{t+1}, \ldots, p_v)$ respectively (note that $r = u - 1$). Then,

$$\begin{aligned} val([\ell, r]) &= \sum_{\substack{j \in [\ell, r]: \\ x_j \in H_0^{t,n}}} val(x_j) \\ &= \sum_{j=t}^{v} val(J_0^j) \\ &\leqslant val(J_0^t) + \left(\sum_{j=t+1}^{v} w_j \right) \\ &\leqslant w_t - \frac{1}{k} + \left(\sum_{j=t+1}^{v} w_j \right) \\ &< \sum_{j=t}^{v} d(j). \end{aligned}$$

where the first inequality is due to the fact that the total value assigned to each player p_j in the right-aligned assignment is at most w_j, the second inequality is due to the choice of p_t, and last inequality holds since $d_j = w_j$ for all players p_j. Thus, Hall's condition is not satisfied for the set of players $p_t, p_{t+1}, \ldots, p_v$ in $\hat{H}$.

Case 2: There are no gaps in the partial assignment. Every item in $N_{[H_0^{t,n}]}(p_t, p_{t+1}, \ldots, p_n)$ has been assigned in the partial assignment. Thus $\bigcup_{j=t}^{n} J_0^j = N_{[H_0^{t,n}]}(p_t, p_{t+1}, \ldots, p_n)$. Once again, let x_ℓ and x_r be the left-most and the right-most items in $N_{[H_0^{t,n}]}(p_t, p_{t+1}, \ldots, p_v)$ respectively. Similarly,

$$\begin{aligned} val([\ell, r]) &= \sum_{j \in [\ell, r]} val(x_j) \\ &\leqslant v(J_0^t) + \left(\sum_{j=t+1}^{n} w_j \right) \\ &\leqslant \left(\sum_{j=t}^{n} w_j \right) - \frac{1}{k} \\ &< \sum_{j=t}^{n} d(j). \end{aligned}$$

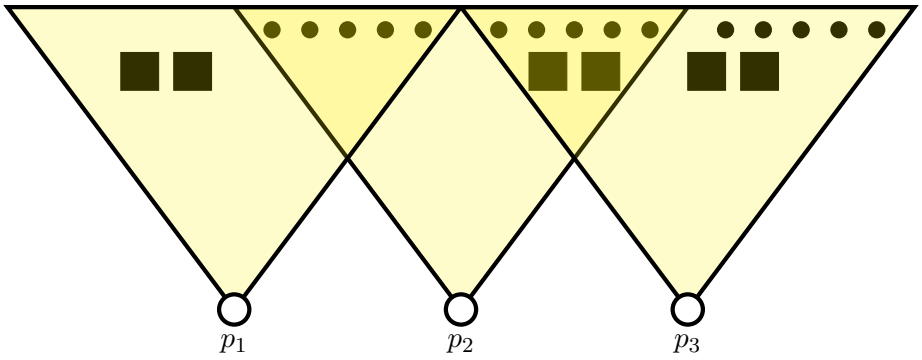


Figure 4.6: Private items can introduce challenges for the dynamic programming scheme. In the figure, circle items are small, with a value of $\frac{1}{10}$, and square items are big, and have a value of $\frac{1}{4}$.

This implies that Hall's condition is violated in H_0 for the set of players p_t, p_{t+1}, ..., p_n. □

Unfortunately, Fact 4.2.2 is not true for the right-aligned assignment restriction, as shown in the following example.

Example 4.2.3. *In the example shown in Figure 4.6, each of the circle items are considered small items and have a value of* $\frac{1}{10}$ *and each of the square items, which are called big items, have a value of* $\frac{1}{4}$*. Now, consider two different right-aligned* 1*-assignments.*

(1) In the first one, player p_3 *gets the 5 rightmost circle items in her neighbourhood as well as the 2 rightmost square items, player* p_2 *also gets the 5 rightmost circle items in her neighbourhood alongside the 2 rightmost square items, and player* p_1 *observes a remainder graph with input vector* $\boldsymbol{\nu} = (5, 2)$*. Fortunately for* p_1*, all these items are in her neighbourhood, so she can be assigned a* 1*-assignment as well and all the players' demands are met.*

(2) In the second assignment, player p_3 *gets all four square items in her neighbourhood. Player* p_2 *has only one choice if she is to be allocated a* 1*-assignment and that choice is to get all the small circle items in her neighbourhood. At the end, player* p_1 *is left with a different remainder graph whose input vector is also* $\boldsymbol{\nu} = (5, 2)$*. This time, the 5 circle items remaining are not in the neighbourhood of* p_1 *(the 5 rightmost circle items), so* p_1 *has to get by with 2 square items which amount to only a* $\frac{1}{2}$*-assignment for her.*

As this example shows, although we have restricted the assignments to right-aligned ones, we still cannot uniquely retrieve the proper (*i.e.*, the one that provides a 1-assignment for all the players) remainder graph from the input vector. Therefore, to avoid storing exponentially many graphs in the table M, we introduce more restrictions on the assignment. These added restrictions ensure that both facts would hold for the final assignment.

Despite the fact that the right-aligned restriction cannot establish a one-to-one mapping between the input vectors and remainder graphs, it takes us one step closer to such mappings.

As illustrated by Example 4.2.3, the reason we could not find an injective relation between the remainder graphs and the input vectors is that in some of the remainder graphs, there exist items that are not assigned to any player, but are also not accessible to any player who has not yet received a bundle of items. For instance, for the second assignment in Example 4.2.3, the 5 rightmost small items appear in the remainder graph, and consequently in the corresponding input vector, but are not in the neighbourhood of p_1. We call such items *private*. More formally, we define private items in the following way.

Definition 23 (Private Items)**.** *For a subgraph $H' = (I', P')$ of the problem instance H, an item $x \in I'$ is called private if it has a degree of 1, that is, it is in the neighbourhood of only one player in H'. For a private item x in H', the player p who is adjacent to x is called the owner of x in H'.*

Definition 24 (Stranded Items)**.** *For a subgraph $H' = (I', P')$ of the problem instance H, an item $x \in H'$ is called stranded if it has a degree of 0, that is, it is in the neighbourhood of no player in H'.*

The private items, if kept unassigned in the graph, can mislead the retrieval procedure in that they appear as available items on the input vector, but become stranded items (their degree becomes zero once the player adjacent to them is removed from the graph, in which case they are wasted). We now define the new restricted assignments.

Definition 25 (Non-wasteful Right-Aligned Assignment)**.** *A right-aligned assignment $\boldsymbol{\mathcal{A}} = (I^1, I^2, \ldots, I^n)$ is called a non-wasteful right-aligned assignment if none of the remainder graphs $H^j = (I \setminus (\bigcup_{t=j+1}^{n} I^t), \{p_1, p_2, \ldots, p_j\})$ contains a stranded item for $j = 1, 2, \ldots, n-1$.*

We first mention the following observation. The proof is straightforward and thus we sketch it briefly.

Observation 3. *Whenever there exists a right-aligned $(1 - \frac{1}{k})$-assignment for a given instance, there also exists a non-wasteful right-aligned $(1 - \frac{1}{k})$-assignment.*

Proof. This is due to the fact that if in any right-aligned assignment, we take the unassigned private items in the remainder graphs and allocate them to their respective owners, we can only increase the value received by each player. Therefore, Fact 4.2.1 holds for non-wasteful right-aligned assignments as well. □

Lemma 5 proves Fact 4.2.2 for this type of restricted assignments.

Lemma 5. *If there exists a non-wasteful right-aligned 1-assignment $\boldsymbol{\mathcal{A}} = (I^1, I^2, \ldots, I^n)$ for an instance of the problem, then there exists a polynomial time algorithm that given the original inclusion-free convex graph H and the input vectors for partial assignments $\boldsymbol{\mathcal{A}}^j = (I^j, I^{j+1}, \ldots, I^n)$ for $j = n-1, n-2, \ldots, 1$ (but not the partial assignments) can*

reconstruct the remainder graphs for every $\boldsymbol{\mathcal{A}}^j$ in such a way that the total value of items in each reconstructed graph is at most $\varepsilon = \frac{2}{k}$ more than the actual corresponding remainder graph for $\boldsymbol{\mathcal{A}}^j$.

Proof. We prove the lemma by providing the reconstruction algorithm. The algorithm is in fact a simple left-to-right sweep[3] of the items. We prove the correctness in Claim 1. For $j = n, n-1, \ldots, 1$, we generate a remainder graph after each partial assignment $\boldsymbol{\mathcal{A}}^j = (I^j, I^{j+1}, \ldots, I^n)$, using only the input vectors. Let $\boldsymbol{\nu} = (\nu_0, v_1, \ldots, \nu_C)$ be the input vector of the remainder graph after this partial assignment. Let $\hat{H} = (\hat{I}, \hat{P})$ denote the remainder graph that our algorithm outputs at the end of this step. We set $\hat{P}$ to be $\{p_1, p_2, \ldots, p_j\}$. As for the set of items $\hat{I}$, for the big items of value q_τ, $\tau = 1, \ldots, C$, we select the left-most ν_τ items in the graph H and add them to $\hat{I}$. For small items, we select a maximal set of left-most items whose total value is less than $\frac{\nu_0+1}{k}$. This method of selection ensures that the items that have been assigned to the players thus far are right-aligned. Furthermore, if at any point, stranded items (of degree 0) appear in the remainder graph, we will return NULL. This ensures that private items are not left unassigned. Note that since $\boldsymbol{\mathcal{A}}$ is assumed to be a non-wasteful right-aligned assignment, so are all its partial assignments as well. Next, we show the correctness of the algorithm by proving Claim 1.

Claim 1. *In a given inclusion-free instance H, for a non-wasteful right-aligned partial assignment $\boldsymbol{\mathcal{A}}^j = (I^j, I^{j+1}, \ldots, I^n)$, the set of big items in $\hat{H}$, the remainder graph reconstructed by the right-to-left sweep algorithm, is identical to that of the original remainder graph of $\boldsymbol{\mathcal{A}}^j$, and the set of small items of $\hat{H}$ is a superset of the set of small items of the original remainder graph. Furthermore, the sum of values for small items in $\hat{H}$ is at most $\frac{2}{k}$ more than that of the original remainder graph.*

Proof. We treat big items and small items separately.

Big items: First, we consider the induced graph on the set of big items of a certain size q_τ for some arbitrary $\tau = 1, \ldots, C$. Assume that $\hat{P} = \{p_1, p_2, \ldots, p_j\}$ for some $1 \leqslant j < n$ (the case where $j = n$ is trivial since the remainder graph would be the original graph H). Also let H' denote the original remainder graph for $\boldsymbol{\mathcal{A}}^j$. For contradiction, assume that the graphs H' and $\hat{H}$ are not identical when induced on the set of big items of value q_τ. Since both graph H' and $\hat{H}$ have the same number of items of value q_τ, there must exist items x' and $\hat{x}$ such that x' appears in H', but not in $\hat{H}$ and $\hat{x}$ belongs to $\hat{H}$, but not to H'. This means that the item $\hat{x}$ is assigned to a player p_x, $x \geqslant j+1$ in one the earlier partial assignments $\boldsymbol{\mathcal{A}}^{j+1}$, $\boldsymbol{\mathcal{A}}j+2, \ldots, \boldsymbol{\mathcal{A}}^n$. The item x' is either adjacent to a player p_y, $1 \leqslant y \leqslant j$, or it is an item of degree 0 in H' and $\hat{H}$ (note that players $p_{j+1}, \ldots, p_n$ do not belong to the graphs H' and $\hat{H}$). If the latter case occurs, it means that x' is a stranded item in H', contradicting the fact that H' is the remainder graph of a non-wasteful right-aligned

[3]Note that the items assigned are aligned to the right, and naturally, the remaining items in the graph would be aligned to the left.

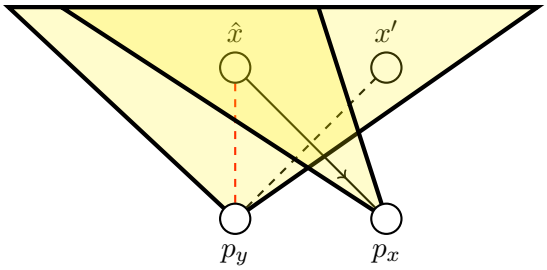


Figure 4.7: If the set of big items of H' and $\hat{H}$ are not identical, a margined inclusion should exist. The arrow indicates assignment in one of the predecessors of H'.

assignment. In fact, when faced with a stranded item, the reconstruction algorithm returns NULL which signals the dynamic programming algorithm to skip the current input vector as it does not correspond to a valid remainder graph. Therefore, we can assume there exists a player p_y adjacent to x'. Since $y \leqslant j$ and $x \geqslant j+1$, we have that $p_y < p_x$. By the adjacency property, we have that $\hat{x} \in N_{[H]}(p_y)$. p_x is not adjacent to x' since otherwise the item $\hat{x}$ assigned to her would be out of alignment. The item x' is assigned to the player p_y, therefore we conclude that $r_{p_x} < r_{p_y}$. Since $p_y < p_x$, it must be the case that $\ell_{p_y} < \ell_{p_x}$. This is a contradiction to the assumption that there are no inclusions in H. This situation is depicted in Figure 4.7.

Small items: Assume that the set of small items of $\hat{H}$ is not a superset of the set of small items of H', meaning that an item x' exists such that x' belongs to H' but not to $\hat{H}$. If there exists another item $\hat{x}$ such that $\hat{x}$ belongs to $\hat{H}$, but not to H', then, similar to the situation in Figure 4.7, we conclude that a margined inclusion should exist in H, which is a contradiction. Therefore, we may assume that $\hat{H}$ is a subgraph of H'. Recall that $\boldsymbol{\nu} = \nu(H') = (\nu_0, \nu_1, \ldots, \nu_C)$ and the sum of values of all the small items in H' is at most $\frac{\nu_0}{k}$ (including x'). The total value of small items in $\hat{H}$ is at least $\frac{\nu_0}{k}$ by definition. This is a contradiction since we assumed $\hat{H}$ is a subgraph of H'. Therefore, the set of small items of $\hat{H}$ must be a superset to that of H'. In the proposed graph $\hat{H}$, we selected the minimal set of small items whose value is greater than or equal to $\frac{\nu_0}{k}$. Such a set may have a value as high as $\frac{\nu_0+1}{k} - \varepsilon$ for some small value of $\varepsilon > 0$ since the value of small items is less than or equal to $\frac{1}{k}$. The minimum total value of small items in H' is no less than $\frac{\nu_0-1}{k}$ by the definition of input vectors, hence the difference of at most $\frac{2}{k}$. This complete the proof of the claim. □

□

Observation 4. *Note that, as mentioned in Claim 1, the algorithm tends to overestimate the value of small items still available in the graph by at most an additive constant factor of $\frac{2}{k}$, but it never underestimates. Therefore when a player p_j expects a bundle of small items worth d_j, she might receive $d_j - \frac{2}{k}$ worth of small items instead*

Based on Observation 4, we next provide a dynamic programming algorithm that enumerates all non-wasteful right-aligned $(1-\frac{3}{k})$-assignments for a rounded instance H. From Lemma 4 and Observation 3 one can conclude that whenever a 1-assignment exists for a given instance of the problem, so does a non-wasteful right-aligned $(1-\frac{1}{k})$-assignment. Therefore, if we only consider non-wasteful right-aligned assignments instead of all assignments, we are still able to find an assignment with objective value at least $(1-\frac{1}{k})$, provided that a 1-assignment for the instance exists in the first place. Lemma 5 and Observation 4 together show that for any non-wasteful right-aligned assignment for an instance of the problem, an approximation of the assignment can be retrieved, and we only loose an additive fraction of $\frac{2}{k}$ in solution quality. Therefore, the general idea of the algorithm is to consider all the m^{C+1} possible input vectors in a dynamic programming algorithm, retrieve a non-wasteful right-aligned $(1-\frac{3}{k})$-assignment for each vector, and mark the feasible one. If it eventually manages to find at least one feasible assignment for p_1 (the last player considered by the algorithm), then it reports success. Before presenting the algorithm, we first introduce two functions that are invoked by the algorithm:

(1) retrieve: Function retrieve takes as input parameters input vector $\boldsymbol{\nu}$ and player index j, and returns remainder graph $H' = (I', P')$. The graph $H' = (I', P')$ that it reconstructs is such that $P' = (p_1, p_2, \ldots, p_j)$ and I' is formed by the left-to-right sweep algorithm in Lemma 5 [4]. Function retrieve returns NULL if at any point during reconstruction, there is at least one stranded item in the remainder graph.

(2) feasible: Function feasible takes as input parameters remainder graph H' and set of items I', and returns either true or false. It returns true if each of the following is true: H' is not NULL, set I' is entirely in the neighbourhood of the last player in H', and the sum of item values in I' is at least $1-\frac{3}{k}$. Otherwise, Function feasible returns false.

Algorithm Assignment (for Inclusion-Free Convex Instances): Algorithm Assignment uses dynamic programming to fill the entries of an $n \times m^{C+1}$ table, in which each row represents a player, and each column represents an input vector (we assume an arbitrary ordering of all valid input vectors). Let the matrix M denote this table. Each entry $(j, \boldsymbol{\nu})$ of M is composed of two fields:

(1) $M(j, \boldsymbol{\nu})$.bit is a binary variable. If $M(j, \boldsymbol{\nu})$ is set to 1, then the input vector $\boldsymbol{\nu}$ is called *marked* for player p_j. Otherwise, the vector $\boldsymbol{\nu}$ is *unmarked* for player p_j.

(2) $M(j, \boldsymbol{\nu})$.ptr either contains NULL or a reference to another input vector $\boldsymbol{\nu}'$ (in the beginning, this pointer is initialized to NULL for every entry).

[4] In fact, Function retrieve also needs the input inclusion-free convex graph H as the third input parameter. To simplify notation, we omit this parameter from the function calls.

Let $\boldsymbol{\nu}_{in} = (\nu_0, \nu_1, \ldots, \nu_C)$ be the input vector for the original graph H. The algorithm starts the forward phase with the last player p_n. For every input graph $\boldsymbol{\nu}$ (*i.e.*, every column of the table M), it forms the remainder graph $\hat{H} = \mathsf{retrieve}(\boldsymbol{\nu}, n-1)$. Let $\hat{I}$ denote the set of items in $\hat{H}$ and also let $I^n = I \setminus \hat{I}$. The set I^n provides a potential assignment for player p_n. The algorithm checks if this assignment is a non-wasteful right-aligned $(1-\frac{3}{k})$-assignment for p_n using the boolean function $\mathsf{feasible}(H, I^n)$. Note that for the first step of the algorithm, the original graph H serves as the remainder graph. We call input vector $\boldsymbol{\nu}$ a *successful* input vector for remainder graph H' and player p_n if p_n is the last player in H' and a call to the function $\mathsf{feasible}(H', I^n)$ returns true. All the successful assignments are marked by setting $M(n, \boldsymbol{\nu}).\mathsf{bit} = 1$. For a successful input vector $\boldsymbol{\nu}$ for player p_n, we set $M(n, \boldsymbol{\nu}).\mathsf{ptr} = \boldsymbol{\nu}_{in}$, while for unsuccessful vectors, the value of $M(n, \boldsymbol{\nu}).\mathsf{ptr}$ is not changed. In this section, we will not use the field ptr in the forward phase of the algorithm.

After this first iteration for player p_n, the algorithm proceeds to a similar procedure for other players. For a player p_j, $1 \leqslant j \leqslant n-1$, it seeks $(1-\frac{3}{k})$-assignments once more. The only difference is that since the assignment should provide a $(1-\frac{3}{k})$-assignment for all players $p_n, p_{n-1}, \ldots, p_j$, for every entry $(j, \boldsymbol{\nu})$ the algorithm looks at all marked entries of the previous row, $j+1$. Let $M(j+1, \boldsymbol{\nu}')$ be such an entry. The algorithm retrieves the graphs $\hat{H} = \mathsf{retrieve}(\boldsymbol{\nu}, j-1)$ and $\hat{H}' = \mathsf{retrieve}(\boldsymbol{\nu}', j)$. We let $\hat{I}$ and $\hat{I}'$ denote the item sets of graph $\hat{H}$ and $\hat{H}'$ respectively. If the set of items $I^j = \hat{I}' \setminus \hat{I}$ is confirmed to be an at least $(1-\frac{3}{k})$-assignment for player p_j by function $\mathsf{feasible}(\hat{H}', I^j)$, the entry $M(j, \boldsymbol{\nu}).\mathsf{bit}$ is set to 1. Furthermore, if $M(j, \boldsymbol{\nu}).\mathsf{ptr}$ is NULL, it is updated to $\boldsymbol{\nu}'$ to mark this step of the assignment, indicating that the input vector $\boldsymbol{\nu}$ was achieved from a remainder graph represented by the input vector $\boldsymbol{\nu}'$. If the function $\mathsf{feasible}$ returns false, the algorithm simply moves to the next marked entry of the previous row. This procedure continues until the algorithms either finds a successful input vector for p_j, or that there is no successful vector in the previous row. In the former case, it reports success and moves to the backward phase in which the actual assignment is retrieved. In the latter case, it simply reports failure. In the backward phase, the non-wasteful right-aligned $(1-\frac{3}{k})$-assignment can be obtained by following the ptr back from the last row to the first. The forward phase is given in Algorithm 3.

Remark 4. *If Function* $\mathsf{feasible}$ *returns* false *at any stage, it may be due to any one of the following three reasons. Either* (i) *the set* I^j *does not have enough total item value, or* (ii) *there are items in set* I^j *which are not in the neighbourhood of player* p_j*, or* (iii) *the remainder graph* $\hat{H}$ *returned by* $\mathsf{retrieve}$ *contains stranded items. In each of these cases, the input vector* $\boldsymbol{\nu}$ *does not represent a valid vector according to the set of restrictions imposed on the assignments. Therefore, the algorithm should ignore* $\boldsymbol{\nu}$ *and simply move to the next input vector.*

Remark 5. *If the* ptr *field is already assigned a value, we do not change it since for the purpose of finding a feasible* $(1-\frac{1}{3})$*-assignments for margined-inclusive free instances, it*

is not important how we arrived at an input vector $\boldsymbol{\nu}$ in the current iteration. If a vector $\boldsymbol{\nu}$ represents an assignment that satisfies players $p_n, \ldots, p_{j+1}$, a simple retrieval procedure that removes items from the right (as suggested in Lemma 5) can retrieve a remainder graph that is a close enough approximation of the true remainder graph.

Remark 6. *In selecting the potential assignment I^j for player p_j at any iteration, we may assign a value in excess of the target value $1 - \frac{3}{k}$. Because we do not wish to leave items stranded or unaligned, the algorithm allocates every item in the difference of two consecutive remainder graphs to the player for whom the iteration is run. Nonetheless, as we formally discuss in Lemma 6, this is not at odds with our purpose of guaranteeing bundles of value $1 - \frac{3}{k}$ or higher in the rounded instance for every player.*

The pseudocode for the algorithm is provided below. In the following, a vector $\boldsymbol{\nu}$ is said to be less than or equal to $\boldsymbol{\nu}'$ if every entry of $\boldsymbol{\nu}$ is less than or equal to its counterpart in $\boldsymbol{\nu}'$.

Algorithm 3 Algorithm Assignment for Max-Min Inclusion-Free Instances

Data: rounded instance H.
Result: returns either a $(1 - \frac{3}{k})$-assignment for all players in the rounded instance, or report failure.

```
for every vector ν in V such that ν ⩽ ν_in do
    Ĥ ← retrieve(ν, n − 1)
    Î ← the set of items in Ĥ ; I^n ← I \ Î
    if feasible(H, I^n) then
        M(n, ν).bit ← 1
        if M(n, ν).ptr == NULL then
            M(n, ν).ptr ← ν_in
        end
    end
end
for j ← n − 1 downto 1 do
    for every vector ν in V such that ν ⩽ ν_in do
        for every vector ν' in V such that ν' ⩾ ν and M(j + 1, ν').bit == 1 do
            Ĥ ← retrieve(ν, j − 1) ; Ĥ' ← retrieve(ν', j)
            Î ← the set of items in Ĥ ; Î' ← the set of items in Ĥ' ; I^j ← Î' \ Î
            if feasible(Ĥ', I^j) then
                M(j, ν).bit ← 1
                if M(j, ν).ptr == NULL then
                    M(j, ν).ptr ← ν'
                end
            end
        end
    end
end
```

In Lemma 6 next, we show that Algorithm Assignment finds a $(1 - \frac{3}{K})$-assignment in a rounded instance if one exists. We use Observation 5 in its proof.

Observation 5. *For a given instance H, assume H' and H'' are two remainder graphs, and that H' is an induced subgraph of H''. Let $\hat{H}' = \mathsf{retrieve}(\nu(H'), j')$ and $\hat{H}'' = \mathsf{retrieve}(\nu(H''), j'')$, where j' and j'' are the indices of the last players in H' and H'' respectively. Then, $\hat{H}'$ is also a subgraph of $\hat{H}''$.*

Proof. The function retrieve preserves the set of players and the set of big items. Therefore, to prove the claim of the observation, it is enough to show that the set of small items in $\hat{H}'$ is a subset of the set of small items in $\hat{H}''$. Let $\boldsymbol{\nu}' = \nu(H')$ be $(\nu'_0, \nu'_1, \ldots, \nu'_C)$ and $\boldsymbol{\nu}'' = \nu(H'')$ be $(\nu''_0, \nu''_1, \ldots, \nu''_C)$. The fact that H' is a subgraph of H'' implies that $\nu'_0 \leqslant \nu''_0$, which means that the function retrieve has not packed more small items from the left to $\hat{H}'$ compared to $\hat{H}''$. Therefore, the set of small items of $\hat{H}'$ is a subset of that of $\hat{H}''$. □

Lemma 6. *Let H be a rounded instance of the problem with n players and m items. The Algorithm Assignment assigns (in polynomial time) to each player a set of items with value at least $1 - \frac{3}{k}$ if a 1-assignment exists for all players. The algorithm returns failure if no $(1 - \frac{1}{k})$-assignment exists for H.*

Proof. From Lemma 4, we know that where there exists a 1-assignment for a rounded instance H, there is also a right-aligned $(1 - \frac{1}{k})$-assignment for the instance, which in turn implies the existence of a non-wasteful right-aligned $(1 - \frac{1}{k})$-assignment. Now, assume the algorithm forms a bundle of items I^j to be potentially assigned to an arbitrary player p_j. The way the algorithm forms the bundles is by finding the difference in the item sets of two remainder graphs, one before the assignment is made and one after. In doing so, it guesses the remainder graphs from their corresponding input vectors. Further assume that the two remainder graphs before and after assignment are H' an H'' respectively, thus H'' is a subgraph of H'. Also, the remainder graphs guessed by the function retrieve are $\hat{H}'$ and $\hat{H}''$. Based on Observation 5, $\hat{H}''$ is also a subgraph of $\hat{H}'$. In a worst case scenario, the function retrieve may guess the total value of small items in $\hat{H}''$ to be at most $\frac{2}{k}$ more than the actual value (the value of small items in H''), while it guesses the same quantity correctly for $\hat{H}'$ (that is, equal to the total sum of small item values in H'). This means that the difference set I^j would have $\frac{2}{k}$ less in small items values than the right-aligned $(1 - \frac{1}{k})$-assignment implied by the original remainder graphs H' and H''. Thus, the set I^j results in a $(1 - \frac{3}{k})$-assignment for player p_j. Therefore, for every possible non-wasteful right-aligned $(1 - \frac{1}{k})$-assignment to players, the algorithm considers a non-wasteful right-aligned $(1 - \frac{3}{k})$-assignment instead. Since the existence of the former results in the existence of the latter, Algorithm Assignment is guaranteed to find a $(1 - \frac{3}{k})$-assignment to players if a $(1 - \frac{1}{k})$-assignment exists. On the other hand, when the algorithm reports failure, we can be certain that a $(1 - \frac{1}{k})$-assignment does not exist, since otherwise our algorithm would be able to find a slightly worse non-wasteful right-aligned $(1 - \frac{3}{k})$-assignment that is guaranteed to exist. □

Now we can prove Theorem 1.

Proof of Theorem 1. Each player receives a set of items with value at least $1 - \frac{3}{k}$ for the rounded instance (see Lemma 6). Each item value is rounded up by at most a multiplicative factor of $1+\frac{1}{k}$. Therefore each player receives a set of items with value at least $(1-\frac{3}{k})/(1+\frac{1}{k}) = 1 - \frac{4}{k+1}$ of the optimal. The two inner for loops of Algorithm Assignment take $m^{2(C+1)}$. The outer loop is run for n players, and finding the remainder graphs and potential bundles need a time of $\mathcal{O}(m+n)$.Thus, the Algorithm Assignment runs in time $\mathcal{O}\left((m+n)nm^{2(C+1)}\right)$. □

4.3 Min-Max Allocation Problem ($R\,||\,C_{\max}$) On Convex Graphs

In this section, we will show that an adaptation of the techniques used in the previous sections, will give us a PTAS for the Min-Max fair allocation of jobs to machines in a Min-Max allocation problem. Most of the techniques used in this section are very similar to those of Section 4.2, therefore we treat the proofs more concisely here, only highlighting the modifications and changes.

4.3.1 Problem Definition and Preliminaries

As in instance of this problem, we are given a set $M = \{M_1, M_2, \ldots, M_m\}$ of identical machines or processors and a set $J = \{J_1, J_2, \ldots, J_n\}$ of jobs. Each job J_j has a same processing time p_j on a subset of machines and it has processing time ∞ on the rest of the machines. The goal is to find an assignment of the (entire set of) jobs to the machines $\boldsymbol{\mathcal{A}} = (J^1, J^2, \ldots, J^m)$, such that the maximum load among all the machines is minimized. Formally, we are given a convex graph $H = (M, J, E)$ where M is a set of machines and J is a set of jobs, and E denotes the edge set. There is an edge in E between a machine and a job if the job can be executed on that machine. Also given is a utility function $p : J \to \mathbb{Q}_{>0}$ which assigns a processing time to each job. With slight abuse of notation, we write $p(J_j)$ as p_j for short, and job J_j requires p_j units of processing time to run to completion on any machine M_i, provided that there exists an edge between J_j and M_i in the set E. If the edge does not exist, the processing time of J_j on M_i is not bounded.

Similarly as before, we assume that the input bipartite graph H satisfies the adjacency and the enclosure properties, *i.e.*, is an inclusion-free convex graph. In other words, we assume that we have an ordering $<_J$ of jobs (which orders the jobs as $J_1, J_2, ..., J_n$) such that each machine can execute consecutive jobs (an interval of jobs). More formally, we denote the interval of job J_i by $[\ell_i, r_i]$. We assume that J_i is before J_j and write $J_i <_J J_j$ whenever $\ell_i < \ell_j$ or $\ell_i = \ell_j$, $r_i \leqslant r_j$.Based on $<_J$, an ordering of jobs that satisfies the adjacency property, we define a lexicographical ordering of the machines in the same manner as Section 4.2. Also as before, we may assume that $0 \leqslant p_i \leqslant 1$ by scaling down the processing time values.

Remark 7. *Similar to the case of Max-Min allocations, if the input convex graph H is inclusion-free, meaning that it satisfies the enclosure property as well as the adjacency property, then not only the neighbourhood of every machines is an interval of jobs, but also every job is adjacent to a consecutive set of machines. More precisely, given the ordering of jobs $<_J$ that satisfies the adjacency and enclosure property and assuming that the set of machines M is ordering lexicographically based on $<_J$, then the neighbourhood of each job $J_j \in J$ forms an interval in the set M. This can be seen as a consequence of Observation 1.*

Hall's Condition: Hall's condition needs to be slightly modified for the case of Min-Max allocations. For subset J' of jobs let $w(J')$ and $w_s(J')$ be the sum of the processing times of all jobs and small[5] jobs in J' respectively. Also let $N_{[H]}(J')$ (or $N(J')$ when the graph H' is implied by context) denote the neighbourhood of the subset of jobs J' in M. A necessary condition for having a maximum load which is at most 1 is that for every subset J' of machines $w(J') \leqslant |N(J')|$. More generally, assume each machine M_i, $1 \leqslant i \leqslant m$, has a maximum *allowable* load denoted by $a(M_i)$. Also, for a subset of machines M', let $a(M')$ be the sum of allowable loads of all the machines in M'. Then, Hall's condition for the Min-Max allocation problem states that in order to have an assignment in which every machine has a load below its allowable maximum, it is necessary to have $w(J') \leqslant a(N(J'))$ for every subset of jobs J'. For the case of Max-Min allocations on inclusion-free graphs, we only need to check this condition for every interval of machines.

Lemma 7. *In order to check Hall's condition for Min-Max allocations in an inclusion-free graph H, it is enough to check the following condition:*

$$\forall \quad [\ell, r\,] \subseteq [1,\, m] : \quad w(\mathcal{J}([\ell,\, r])) \leqslant \sum_{i=\ell}^{r} a(M_i) \tag{4.3.1}$$

in which $\mathcal{J}([\ell,\, r])$ denotes the set of jobs whose entire neighbourhood of machines falls in the interval $[\ell,\, r]$. We refer to Condition 4.3.1 as Hall's condition for Min-Max allocations.

Proof. The proof mostly follows the proof of Lemma 3 except for the fact that it uses the property mentioned in Remark 7. Assume we are given an inclusion-free convex graph H. As before, we show that there exists a subset of the jobs J' for which *(i)* $N(J')$ is not an interval and therefore can be represented as the union of several maximal intervals, and *(ii)* Hall's condition is violated if and only if there exists an interval of machines $[\ell,\, r]$ for which Condition 4.3.1 is violated.
$\Rightarrow$: Assume that $w(J') > a(N(J'))$ and that there exist several maximal intervals of machines $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_t$ whose union gives $N(J')$. Since each job is adjacent to an interval of machines by Remark 7, there exists a corresponding partition of J' into subsets $J'_1, J'_2, \ldots, J'_t$, such that $N(J'_i) = \mathcal{N}_i$. Since Hall's condition is violated, we have

[5]We use definitions for *small* and *big* jobs similar to the definitions for *small* and *big* items in Section 4.2.2.

$$a(N_1) + \ldots + a(N_t) = a(N(J')) < w(J') = w(J'_1) + \ldots + w(J'_t)$$

Thus, there must exist an $i, 1 \leqslant i \leqslant t$, for which $a(\mathcal{N}_i) = a(N(J'_i)) < w(J'_i)$. Let ℓ and r be the leftmost and rightmost machines in $N(J'_i)$. We have that $a(N(J'_i)) = \sum_{i=\ell}^{r} a(M_i) < w(J'_i) \leqslant w(\mathcal{J}([\ell, r]))$ since $w(J'_i) \subseteq w(\mathcal{J}([\ell, r]))$.
$\Leftarrow$: Assume that Condition 4.3.1 is violated for an interval of machines $[\ell, r] \subseteq [1, m]$, meaning that $w(\mathcal{J}([\ell, r])) > \sum_{i=\ell}^{r} a(M_i)$. Let J' be $\mathcal{J}([\ell, r])$. Then, $N(J') \subseteq [\ell, r]$ since otherwise it means there exists a job in $J' = \mathcal{J}([\ell, r])$ whose neighbourhood stretches beyond the interval $[\ell, r]$, contradicting the definition of $\mathcal{J}([\ell, r])$. Therefore $w(J') > \sum_{i=\ell}^{r} a(M_i) \geqslant a(N(J'))$. Therefore, Hall's condition is violated for the set J'. □

Rounding the Instance

Before running the algorithm on the instance, we round the processing times of jobs based on an input error parameter k. First of all, by properly scaling the values we assume every job has a processing time less than or equal to one. We say a job is *small* if its processing time is less than or equal to $\frac{1}{k}$ after scaling, and we say a job is *big* if its processing time is strictly larger than $\frac{1}{k}$. As opposed to the Max-Min case, we round down the processing times for big jobs for the Min-Max case. Thus, if p_j, the processing time of job J_j, is in the interval $[\frac{1}{k}(1+\frac{1}{k})^i, \frac{1}{k}(1+\frac{1}{k})^{i+1})$, then it is replaced by $\frac{1}{k}(1+\frac{1}{k})^i$. Using this method, we obtain at most $C = \lceil \frac{\log k}{\log(1+\frac{1}{k})} \rceil$ distinct processing times for big jobs (each of which is more than $\frac{1}{k}$, the maximum processing time for small jobs). For $1 \leqslant \tau \leqslant C$, let $q_\tau = \frac{1}{k}(1+\frac{1}{k})^\tau$. Each q_τ, $1 \leqslant \tau \leqslant C$, denotes the rounded processing time of category τ of big jobs.

4.3.2 The Algorithm for Inclusion-Free Convex Graphs

The main theorem of this section is the following.

Theorem 2. *Let H be an instance of the problem before rounding with n jobs and m machines. Then, for $k \geqslant 4$ there exists a $(1 + \frac{4}{k} + \frac{3}{k^2})$-approximation algorithm for the Min-Max allocation problem on inclusion-free convex graphs with the running time of $\mathcal{O}\left((m+n)mn^{2(C+1)}\right)$ in which $C \leqslant k^{1.4}$.*

To show this result, we borrow from the techniques used in Section 4.2 extensively. We use the notions of *t-assignment*, *input vector*, *assignment vector*, and *right-aligned* assignments in the same sense as we did for the Max-Min allocations (jobs are allocated to machines here, instead of items to players). We modify the definition of input vectors slightly for Min-Max allocations below.

Definition 26 (t-assignment for the Min-Max case)**.** *A t-assignment, for any $t \geqslant 0$, is a feasible assignment such that every machine M_i receives a set of jobs $J^i \subseteq [\ell_{M_i}, r_{M_i}]$ with total processing time at most t.*

Definition 27 (Input Vector)**.** *For a given convex graph H for an input instance with ν_τ big jobs of processing time q_τ for $\tau = 1, 2, \ldots, C$ and small jobs of total processing time in the interval $[\frac{\nu_0}{k}, \frac{\nu_0+1}{k})$, an input vector is a configuration vector of the form $\nu(H) = \boldsymbol{\nu} = (\nu_0, \nu_1, \ldots, \nu_C)$.*

Then, we can prove the following *alignment* lemma.

Lemma 8 (The Alignment Lemma for Min-Max Allocations)**.** *Suppose there exists a 1-assignment for a given convex graph H. Then, there also exists a <u>maximal</u> $(1 + \frac{1}{k})$-right-aligned assignment for H. Furthermore, the two assignments will have an identical assignment vector.*

Proof. We assign the jobs in two rounds and make use of the adjacency property. During the first round, the big jobs are aligned to the right in the same manner as Lemma 4. In doing so, we do not change the assignment vector since the same number of big jobs as the 1-assignment are allocated to the machines. During the second round, we assign the small items. The only difference between the assignment of jobs and the assignment of items is that we would prefer to allocate as many jobs as possible to a machine as long as the make-span is below the threshold of $1 + \frac{1}{k}$. Therefore, we pack maximal sets of small jobs and assign them to the machines. Due to the way the assignment vectors are defined, we still have not changed the vector after this phase. □

By the problem definition, we are not allowed to have stranded jobs in the Min-Max allocation problem. Therefore, every right-aligned assignment that we make is also a *non-wasteful* right-aligned assignment. The following two facts hold for any non-wasteful right-aligned assignment of jobs to machines.

Fact 4.3.1. *Whenever there exists an arbitrary 1-assignment of jobs to machines for an instance of the problem, there also exists a restricted $(1 + \varepsilon)$-assignment, in the sense that all the jobs assigned are restricted to be aligned to the right.*

Fact 4.3.2. *Given that a 1-assignment exists for an instance of the problem, there also exists a polynomial time algorithm that, given the input vectors for every step (partial assignment) of a $(1 + \varepsilon)$-assignment, reconstructs the remainder graph of every step in such a way that the total processing time of the jobs in each reconstructed remainder graphs is*

- *exactly the same as that of the corresponding remainder graph in the original (and unknown) 1-assignment for every category $\tau = 1, 2, \ldots, C$ of big jobs and,*
- *only a small fraction ε less than its counterpart in the original remainder graph for small jobs.*

While Lemma 8 verifies Fact 4.3.1 for non-wasteful right-aligned assignments, the following lemma ensures that Fact 4.3.2 also holds for these types of assignments.

Lemma 9. *If there exists a non-wasteful right-aligned* 1*-assignment* $\boldsymbol{\mathcal{A}} = (J^1, J^2, \ldots, J^m)$ *for an instance of the problem, then there exists a polynomial time algorithm that, given the original inclusion-free convex graph* H *and the input vectors for partial assignments* $\boldsymbol{\mathcal{A}}^j = (J^j, J^{j+1}, \ldots, J^m)$ *for* $j = m-1, m-2, \ldots, 1$ *(but not the partial assignments), can reconstruct the remainder graphs for every* $\boldsymbol{\mathcal{A}}^j$ *in such a way that the total processing time of jobs in each reconstructed graph is at most* $\varepsilon = \frac{2}{k}$ *less the actual corresponding remainder graph for* $\boldsymbol{\mathcal{A}}^j$*.*

Proof. The proof uses a left-to-right sweep again. We do not provide the details here since it is identical to the proof of Lemma 5, except for the way small jobs are chosen in the reconstruction process. For small jobs, we select a minimal set of left-most jobs whose total processing time in greater than $\frac{\nu_0 - 1}{k}$ and add them to the remainder graph. Since the processing time of each small job is less than $\frac{1}{k}$, this ensures that the sum of processing times of the jobs we choose in the reconstructed remainder graph is less than or equal to that of the original remainder graph, but the deficit is not more than $\frac{1}{k}$. The rest of the proof follows the proof of Lemma 5. □

Algorithm Assignment (for the Min-Max Problem on Inclusion-Free Convex Instances): Algorithm Assignment for the Min-Max allocation problem is a dynamic programming algorithm similar to the algorithm given in Section 4.2. Again we make use of two utility functions, namely the reconstruction function retrieve and the boolean function feasible. Function retrieve reconstructs a remainder graph based on an input vector and a player index which potentially has slightly less total processing time than the original remainder graph of a partial assignment, and function feasible checks whether a given assignment of jobs is feasible in a remainder graph and does not leave any stranded jobs behind. Using these functions, the algorithm fills in the entries of a $m \times n^{C+1}$ table T. Each entry $(i, \boldsymbol{\nu})$ of the table has two fields:

(1) $T(i, \boldsymbol{\nu})$.bit: this binary value is set to 1 if the input vector $\boldsymbol{\nu}$ is marked for machine i, and set to 0 otherwise.

(2) $T(i, \boldsymbol{\nu})$.ptr: contains either NULL or a reference to another input vector $\boldsymbol{\nu}'$ in the previous row, intended for use in the backward phase of the algorithm where an actual assignment of the jobs to the machines is retrieved from the table T.

The algorithm is given in Algorithm 4

Lemma 10. *Let* H *be a rounded instance of the problem with* n *jobs and* m *machines. Algorithm Assignment assigns (in polynomial time) to each machine a set of jobs with value at most* $1 + \frac{3}{k}$ *if a* 1*-assignment exists for all machines. The algorithm returns failure if no* $(1 + \frac{1}{k})$*-assignment exists for* H*.*

Algorithm 4 Algorithm Assignment for Min-Max Inclusion-Free Instances

Data: rounded instance H.

Result: returns either a $(1 + \frac{3}{k})$-assignment for all machines in the rounded instance, or report failure.

```
for every vector ν in 𝒱 such that ν ⩽ ν_in do
    Ĥ ← retrieve(ν, m − 1)
     Ĵ ← the set of jobs in Ĥ ; J^m ← J \ Ĵ
     if feasible(H, J^m) then
        T(m, ν).bit ← 1
         if T(m, ν).ptr == NULL then
            T(m, ν).ptr ← ν_in
        end
    end
end
for i ← m − 1 downto 1 do
    for every vector ν in 𝒱 such that ν ⩽ ν_in do
        for every vector ν' in 𝒱 such that ν' ⩾ ν and T(i + 1, ν').bit == 1 do
            Ĥ ← retrieve(ν, i − 1) ; Ĥ' ← retrieve(ν', i)
             Ĵ ← the set of jobs in Ĥ   ;   Ĵ' ← the set of jobs in Ĥ' ; J^i ← Ĵ' \ Ĵ
             if feasible(Ĥ', J^i) then
                T(i, ν).bit ← 1
                 if T(i, ν).ptr == NULL then
                    T(i, ν).ptr ← ν'
                end
            end
        end
    end
end
```

Proof. By Lemma 8, a $(1+\frac{1}{k})$-assignment exists whenever there is a 1-assignment for the problem instance. Using the retrieve function, we may guess the remainder graph of any of the partial assignments to have a deficiency in total processing time which is no more than $\frac{2}{k}$ when compared to the original remainder graph. This together with the error caused by the alignment imply the total sum of processing times of the jobs assigned to each machine is at most $\frac{3}{k}$ more than the same value in the 1-assignment (if a successful assignment is identified). It is also straightforward to see that whenever there exists no $(1+\frac{1}{k})$-assignment for the instance, the algorithm fails to recover a feasible assignment for a machine, resulting in a failure report. This completes the correctness proof of the Algorithm Assignment. □

In the last part, we prove Theorem 2 using the lemmas in this section.

Proof of Theorem 2. First we discuss the approximation guarantee. Through alignment and reconstruction of the remainder graphs, we may pack an extra $\frac{3}{k}$ units of processing time in the bundle of jobs assigned to a machine. This extra value is magnified by a multiplicative factor of $1+\frac{1}{k}$ in the rounded instance. Therefore, the approximation guarantee would be $(1+\frac{3}{k})\cdot(1+\frac{1}{k}) = 1+\frac{4}{k}+\frac{3}{k^2}$. We now discuss the running time of the algorithm. The two inner loops of the Algorithm Assignment take a combined total time of $n^{2(C+1)}$ while the outer loop runs once for each machine for a total of m times. Retrieving the remainder graph can be done in $\mathcal{O}(m+n)$, hence the running time is $\mathcal{O}\left((m+n)mn^{2(C+1)}\right)$ in total. The correctness is already shown in Lemma 10. □

4.4 Extensions to Other Ordered Instances

In this section, we discuss other types of ordered instances solvable using a similar dynamic programming technique as in the previous sections. The ordered instances we consider here can be regarded as an extension of the notion of *laminar families*. To explain the results, we first introduce the laminar families of sets and present a PTAS for the $R\,||\,C_{\max}$ problem over ordered instances defined by them. Then, we define an extension of the laminar families and show how to apply the method for them.

4.4.1 Laminar Families of Sets

Definition 28 (Laminar Family of Sets)**.** *A family $\mathcal{L}$ of sets over a ground set of elements $U = \{e_1, e_2, \ldots, e_n\}$ is called a laminar family of sets if for every two distinct members L_1 and L_2 we have one of the following conditions:*

- $L_1 \cap L_2 = \varnothing$
- $L_1 \subseteq L_2$
- $L_2 \subseteq L_1$

For any two members of a laminar family, either they are disjoint, or one is a subset of another. Now, consider an instance of a restricted $R||C_{\max}$ problem given by the set of machines $M = \{M_1, M_2, \ldots, M_m\}$ and the set of jobs $J = \{J_1, J_2, \ldots, J_n\}$. In this section, we provide a PTAS for the problem when the family of neighbourhoods of the machines, $\{N(M_1), N(M_2), \ldots, N(M_m)\}$, forms a laminar family over the set of jobs J. As a reminder, we should mention that in the restricted $R||C_{\max}$ problem we are also given a bipartite graph $H = (M, J, E)$. A job J_j has a fixed processing time p_j for all the machines connected to it in H and a processing time of ∞ for the rest. In this sense, the neighbourhoods of machines show the set of jobs that they are capable of processing.

Laminar families of sets are known to represent hierarchies among sets. We can depict these hierarchies in the form of forests. Example 4.4.1 shows one such hierarchical structure.

Example 4.4.1. *Consider the ground set to be the set of eight jobs $J = \{J_1, J_2, \ldots, J_8\}$. Also, consider a set of eight machines with the following neighbourhoods:*

- $N(M_1) = \{J_1, J_2, J_3, J_4, J_5, J_6, J_7, J_8\}$
- $N(M_2) = \{J_1, J_2, J_3, J_4, J_5, J_6\}$
- $N(M_3) = \{J_7\}$
- $N(M_4) = \{J_4\}$
- $N(M_5) = \{J_1, J_2, J_3\}$
- $N(M_6) = \{J_5\}$
- $N(M_7) = \{J_1\}$
- $N(M_8) = \{J_2\}$

Every two neighbourhoods follow the rules of laminar families. Any neighbourhood that does not contain the neighbourhood of any other machine represents a leaf in this forest. For this example, the leaves are the neighbourhoods of M_3, M_4, M_6, M_7, and M_8. Now we can remove the leaves from the family and recursively complete every tree in this forest in a bottom-up fashion. Figure 4.8 depicts the tree for the neighbourhoods above.

The main theorem in this section is Theorem 3.

Theorem 3. *Let H be an instance of the problem before rounding with n jobs and m machines. Then, for $k \geqslant 2$ there exists a $(1 + \frac{2}{k} + \frac{1}{k^2})$-approximation algorithm for the Min-Max allocation problem when the neighbourhoods of the machines form a laminar family. The running time is $\mathcal{O}\left(m(n+1)n^{2C+2}\right)$ in which $C \leqslant k^{1.4}$.*

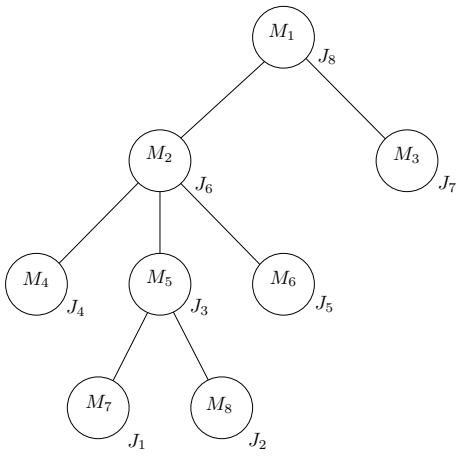


Figure 4.8: A Hierarchical Representation of a Laminar Family

We provide a dynamic programming algorithm to prove Theorem 3 [6]. Assume, as before, we have guessed a target value t for the makespan on every machine. Our objective is to find a t-assignment as defined in Section 4.3.2. That is, assign jobs to the machines in such a way that the makespan for every machine is below t. We use the same rounding method as in Section 4.3.1. The processing times are also normalized, so we are interested in a 1-assignment rather than a t-assignment. We also assume that we are given the forest representation of the neighbourhoods. $\mathcal{F}$ denotes this representation. Every vertex of $\mathcal{F}$ is a machine whose neighbourhood in the set J contains the neighbourhoods of all other machines in its subtree. Note that we can create the forest in polynomial time from the neighbourhoods. We handle each tree of the forest separately. We first describe the intuition behind the algorithm and then, provide the detail.

We first draw a comparison between the PTAS for Min-Max problem on bipartite permutation graphs presented in Section 4.3.2 and the one we show in this section for laminar families of machine neighbourhoods. For convex graphs, we need to take certain measures for assigning the jobs to ensure we can recreate a close approximation of the remainder graphs later on. For instance, we start the assignment process from the right-most player and only consider non-wasteful right-aligned assignments. The PTAS for laminar families differs from the one for convex graphs in the way remainder graphs are treated. If we carry on the allocations in a particular order, storing remainder graphs (or retrieving them based on a given input vector) becomes redundant. We explain this further in Observation 6.

Observation 6. *To retrieve a remainder graph that corresponds to a given input vector* $\boldsymbol{\nu} = (\nu_0, \nu_1, \ldots, \nu_C)$*, we can select any arbitrary remainder graph that respects the vector* $\boldsymbol{\nu}$*. By respecting the vector we mean the number of big items in the graph should match* ν_τ*'s*

[6] An algorithm of a similar nature is due to Schwarz [99].

for $\tau = 1, 2, \ldots, C$ and the total value of small items rounded down to the closest integer multiple of $\frac{1}{k}$ should be $\boldsymbol{\nu}_0$.

Proof Sketch. Consider a leaf machine w and assume we have identified a set of input vectors $\{\boldsymbol{\nu}_1, \boldsymbol{\nu}_2, \ldots, \boldsymbol{\nu}_t\}$ for $t \leqslant m^{C+1}$ that admit a $(1 + \frac{1}{k})$-assignment for w. As a reminder, these vectors indicate the number of remaining items after making a successful $(1 + \frac{1}{k})$-assignment to w (we use the terms "successful assignment" and "successful input vector" in the same sense as in previous sections). For any vector $\boldsymbol{\nu}_i$, let $\boldsymbol{\alpha}_i = \boldsymbol{\nu}_{in_w} - \boldsymbol{\nu}_i$ be the vector corresponding to the $(1 + \frac{1}{k})$-assignment. The assignments of jobs to w only affect the predecessors of w in the tree. By definition of laminar families, all the predecessor of w have access to all items in the neighbourhood of w (and possibly more). Therefore, the order by which the jobs are assigned to w does not matter as long as the number of jobs matches those indicated in an $\boldsymbol{\alpha}_i$. Therefore, one can choose any remainder graph that respects the corresponding input vector[7]. Inductively, we remove the leaves once we find their successful assignments and continue the process on the reduced tree.

Algorithm Assignment (for the Min-Max Problem on Laminar Families): The dynamic programming algorithm receives the root node of a tree from the forest as input and visits the players (machines) in a depth-first order. It then fill the elements of a matrix $T(j, \boldsymbol{\nu})$, for every $j \in J$ and every vector $\boldsymbol{\nu}$ in $\mathcal{V}$. The matrix T indicates the successful assignments and is the return value of the algorithm. It is a simplified version of the matrix in Section 4.3.2 as it is simply a zero-one matrix. We do not make the use of the `ptr` field since, as explained in Observation 6, the retrieval of remainder graphs for laminar families is straightforward. Let $\mathbf{0} = (0, 0, \ldots, 0) \in \mathcal{V}$ be the input vector specifying no available jobs. At the end of the computation, if $T(r, \mathbf{0})$ is 1, the algorithm returns the matrix T and indicates a successful assignment to the root node r (hence recursively, to all the machines) that leaves no jobs behind. If $T(r, \mathbf{0})$ is 0, it returns failure.

For every node w of the tree, we store an array $U(w, \boldsymbol{\nu})$ for all $\boldsymbol{\nu} \in \mathcal{V}$. U is a zero-one matrix with rows corresponding to the machines and columns corresponding to vectors in $\mathcal{V}$. For a given machines w, the corresponding row in U indicates every possible input vector (or equivalently, the number of items that may be available) after we make some successful assignments to the children of w. We also keep an input vector $\boldsymbol{\nu}_{in_w}$ for every node w. Note that the rows of both matrices T and U are indexed by the nodes of the tree.

The main idea of the algorithm is first to find the possible input vectors for every node after the assignments for the children are computed recursively. Then, for every possible input vector, iterate over all assignments vectors, find the successful ones and mark the corresponding entries of the matrix T. We now discuss how the algorithm fills the entries of U. For a leaf node w, the only 1 in the w^{th} row of U is in $U(w, \boldsymbol{\nu}_{in_w})$. For an intermediate

[7] Alon et al. use this fact in their PTAS for the case of complete graphs [2] .

node w, $U(w, \boldsymbol{\nu}_{in_w})$ is the first entry we set to 1. Note that not assigning any jobs to the children of w is still a successful assignment (although it wastes the computational power of some machines, it keeps their makespan below the target). Then, we consider the children of w one by one in a recursive function call, find their successful input vectors, and mark them in the matrix T. Based on the marked entries of T, we can update the bit vector $U(w, \boldsymbol{\nu})$ of all possible input vectors for w. This is done in the function aggregate. In the next paragraphs, we explain all the utility functions called by the Algorithm Assignment.

- aggregate: The algorithm calls this function only for intermediate nodes of the tree. After recursively finding successful input vectors of the children (stored in T), the function aggregate first computes the assignment vectors corresponding to the successful assignments by comparing the input vector of the child with vectors marked in T. Then, it deducts the assignment vectors from the possible input vectors of U. Note that at the beginning, the only possible input vector stores in U for a machine w is $\boldsymbol{\nu}_{in_w}$, which assumes the availability of all jobs in the neighbourhood of w. As successful assignments to children are found, the algorithm gradually expands the set of possible input vectors by marking entries of U. (see Algorithm 6).

- feasible: This is a simpler version of the feasible function we used in Section 4.3.2. Since we do not store the remainder graphs anymore, we only need to check whether the total processing time for the set of jobs J does not exceed $1 + \frac{1}{k}$. If so, the function feasible returns true. Otherwise, it returns false.

- parent: Receives a node (machine) w and returns its parent node in the tree.

After updating the matrix U, we find successful assignments for every possible input vector of w marked in U (the two nested **for** loops). This is done by checking every potential assignment vector $\boldsymbol{\alpha} = \boldsymbol{\nu} - \boldsymbol{\nu}'$ where $\boldsymbol{\nu}$ is a possible input vector for w before the assignment and $\boldsymbol{\nu}'$ is a possible remainder vector after the assignment. Let the vector $\boldsymbol{\alpha}$ be $(\alpha_0, \alpha_1, \ldots, \alpha_C)$. For each category τ, $1 \leqslant \tau \leqslant C$, of big jobs in $\boldsymbol{\alpha}$, we choose a set of big items of size q_τ and cardinality α_τ. For the small jobs, we greedily add them to the set J as long as their total processing time is strictly less than $\frac{\alpha_0}{k}$. Since the processing time for every small job is at most $\frac{1}{k}$ this does not increase the makespan by more than $\frac{1}{k}$ from 1. Eventually, a successful assignment for the instance should be able to assign all the items while keeping the makespan below $1 + \frac{1}{k}$. Since the root of the tree is adjacent to all jobs, it can select any jobs left behind by its children, subject to the makespan constraint. Therefore, at the end of the computation, the algorithm checks $T(r, \mathbf{0})$. This entry indicates if an assignment of jobs to the root node r and all its children exists that **i)** respects the makespan constraint, and **ii)** leaves no items unassigned. If so, success is returned.

Now, we prove the correctness of the Algorithm Assignment.

Algorithm 5 Algorithm Assignment for Laminar Families

Data: rounded instance H, the root node r for a tree in $\mathcal{F}$.
Result: returns either a $(1 + \frac{1}{k})$-assignment for all machines in the rounded instance, or report failure.

```
let ν_{in_r} denote the input vector for r
U(r, ν_{in_r}) ← 1
for every child w of r do
    let ν_{in_w} be the input vector for w
    recursively run Algorithm Assignment(H, w) and update T
    aggregate(U, T, r)
end
for every vector ν in V such that U(r, ν) == 1 do
    for every vector ν' in V such that ν' ⩽ ν do
        J ← the set of jobs in ν − ν'
        if feasible(J) then
            T(r, ν') ← 1
        end
    end
end
if parent(r) == NULL ∧ T(t, 0) == 0 then
    return NULL
end
return T
```

Algorithm 6 aggregate Function Called by Algorithm Assignment

Data: matrix U, matrix T, a node w
Result: updates the w^{th} row of matrix U based on the same row in T.

```
for every vector ν ∈ V for which T(w, ν) == 1 do
    α ← ν_{in_w} − ν
    for every vector ν' ∈ V for which U(w, ν') == 1 do
        ν'' ← ν' − α
        U(w, ν'') ← 1
    end
end
```

Lemma 11. *Assume given is an instance of the $R||C_{\max}$ problem on laminar families that admits a* 1*-assignment. Algorithm Assignment (Algorithm 5) reports success if and only if it finds an assignment that assigns jobs to the machines such that i) the maximum makespan among all machines is* 1*, ii) all jobs are assigned.*

Proof. It is straightforward to see if the algorithm reports success, then the assignment found by the algorithm satisfies the two conditions of the lemma. This is true because it only marks the input vectors corresponding to assignments that have a makespan below the threshold of $1 + \frac{1}{k}$ and it only reports success if all jobs are assigned.

For the other direction, we use induction. For a leaf node w, if a 1-assignment exist, then so there is a $(1 + \frac{1}{k})$-assignment. The algorithm checks all possible vectors for such assignments and finds one as a successful one. Now, assume w is an intermediate node of the tree. Inductively assume that all assignments satisfying the two conditions have been found for the children of w. By Observation 6, the jobs in these assignments can be selected arbitrarily from the neighbourhoods of the children, not affecting the assignment to w. So, the algorithm only needs to look at all possible ways of combining the assignments to children, find out what the input vectors for the set of available items for w are after each combination, and then look for a successful assignment as if w is a leaf node. The total number of input vectors for all possible ways to combine these assignments is polynomial as it is no more than m^{C+1}. The function aggregate finds all these vectors. Then, the algorithm can detect $(1 + \frac{1}{k})$-assignments for w in a similar way it does for the leaves. □

Proof of Theorem 3. The correctness is shown in Lemma 11. We discuss the time complexity here. The algorithm performs a depth-first search on the tree of machines. For each node, Algorithm Assignment is called once and the function aggregate is also called once. Therefore, the running time would be $m \cdot (\mathcal{O}(\text{time complecity of Algorithm Assignment}) + \mathcal{O}(\text{time complexity of aggregate}))$. The major time of Algorithm Assignment for a node is spent in the two nested **for** loops. The time complexity for them is $n^{2(C+1)} \times \mathcal{O}(\text{feasible})$. The function feasible scans the jobs at most once, so it time complexity is $\mathcal{O}(n)$. The function aggregate has a pair of nested **for** loops as well. The running time of them is $\mathcal{O}\left(n^{2(C+1)}\right)$. The total running time of the algorithm is $\mathcal{O}\left(m \cdot (n \cdot n^{2(C+1)} + n^{2(C+1)})\right) = \mathcal{O}\left(m(n+1)n^{2C+2}\right)$. As for the approximation factor, the algorithm finds a $(1 + \frac{1}{k})$-assignment for the rounded case. We must account for the fact that the processing times have been rounded down. Therefore, the approximation guarantee of the algorithms is $(1+\frac{1}{k}) \times (1+\frac{1}{k}) = 1+\frac{2}{k}+\frac{1}{k^2}$. □

4.4.2 Extended Laminar Families of Sets

We define the "extended laminar family" of sets as follows:

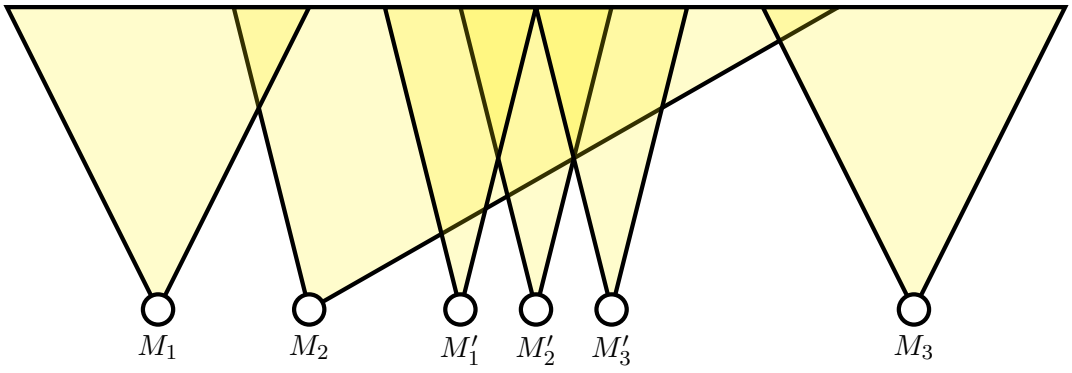


Figure 4.9: Extended laminar families of neighbourhoods.

Definition 29 (Extended Laminar Family of Sets)**.** *A family $\mathcal{L}$ of sets over a ground set of elements $U = \{e_1, e_2, \ldots, e_n\}$ is called an extended laminar family of sets if for every two distinct members L_1 and L_2 we have one of the following conditions:*

- $L_1 \cap L_2 = \varnothing$
- $L_1 \subseteq L_2$
- $L_2 \subseteq L_1$
- $L_1 \setminus L_2 \neq \varnothing$ *and* $L_2 \setminus L_1 \neq \varnothing$, L_1 *and* L_2 *satisfy the enclosure property, and for every* $L' \subseteq L_1$ *and every* $L'' \subseteq L_2$ *we have that* $L' \cap L'' = \varnothing$.

First, note that the extended laminar family of sets has the laminar family as its special case. If no two sets L_1 and L_2 are of the form of the last condition, then family becomes laminar. In this section, we assume that the neighbourhoods of the machines from an extended laminar family. An instance of this problem is shown in Example 4.4.2.

Example 4.4.2. *As shown in Figure 4.9, machines M_1, M_2, and M_3 overlap each other while satisfying the enclosure property. A sequence of machines, M'_1, M'_2, and M'_3 have their neighbourhoods entirely contained in the neighbourhood of M_2. They also overlap and satisfy the enclosure property. But note that they do not intersect with any of the neighbourhoods of M_1, M_3, or any other neighbourhood entirely within $N(M_1)$ and $N(M_3)$.*

Similar to laminar families, extended laminar families can also be represented with forests. The main difference is that in a forest representation of an extended laminar family of sets, each node does not necessarily map to a single machine. Instead, it may represent a sequence of the machines that overlap and satisfy the enclosure property. Lemma 12 shows that the recognition algorithm for this class of instances is polynomial time.

Lemma 12. *Given a bipartite graph $H = (X, Y, E)$, there is an algorithm that, in polynomial time, can determine whether or not the neighbourhoods of the vertices of X in Y form an extended laminar family of sets.*

Proof. For any two vertices x_1 and x_2, $N(x_1)$ and $N(x_2)$ satisfy the adjacency property. The reason is that either $N(x_1)$ and $N(x_2)$ do not intersect at all, in which case they respect the adjacency property, or they intersect in one of the two ways: **i)** one is entirely contained in the other, in which case they satisfy the adjacency property although they violate the enclosure property, or **ii)** they partially intersect, in which case they satisfy the enclosure property by the definition of the extended laminar families. Therefore, the entire graph H satisfies the adjacency property and is a convex graph. There exist a linear recognition algorithm for the convex graph that can produce the intervals [17]. We run the algorithm on the graph H. If it rejects it as a convex graph, then it cannot be an extended laminar family of sets. Therefore we reject it too. If the algorithm accepts the graph H, then it returns the neighbourhoods as intervals. We iterate over vertices of X in the lexicographical ordering we defined for convex graphs in Section 4.2.1. For a vertex x, we find the right-most point of any interval entirely inside $N(x)$. Let r show the corresponding index. We also find the left-most point of any interval partially overlapping $N(x)$ whose left end is not to the right of the right end of $N(x)$ (comes after x in the lexicographical ordering). Let ℓ denote this index. If $\ell < r$, then we stop and report rejection. Otherwise, we continue to the next vertex x in the ordering. If we do not reject by the last vertex in X, we report acceptance. It is straightforward to see that the algorithm rejects the graph H only if there is a violation of the four conditions of the extended laminar families. Also, the calculations of r and ℓ can be done in time linear in $|\,Y\,|$. Therefore, the algorithm runs in polynomial time. □

According to Lemma 12, we can identify extended laminar families among the neighbourhoods of the machines. With minor modifications to the recognition algorithm, we can also get the forest representation. In this representation, each node may indicate a set of machines whose overlapping neighbourhoods satisfy the adjacency property. We call such a set the *batch* of a node. For a machine M_i, the set of children of M_i is a minimal set of machines whose neighbourhoods are entirely within that of M_i. By minimal set, we mean minimal subject to inclusion. Therefore if $N(M_k) \subset N(M_j)$ and $N(M_j) \subset N(M_i)$, we only count M_j as a child of M_i, with M_k being a grandchild of M_i. Based on the forest representation, we provide a PTAS in Algorithm 7.

Remark 8. *Note that for the laminar families of sets, every node of the tree w represents a machine. Therefore we could use the label w both to refer to the node and to the machine it represents. In the extended laminar family setting, a node represents a batch of machines. Therefore, we use labels to refer to either the nodes of the tree or batches of the machines, while reserving M for the individual machines.*

Algorithm Assignment (for the Min-Max Problem on Extended Laminar Families): The algorithm resembles Algorithm 5 for the most part. The difference is in finding successful assignments for the nodes once the assignments for children are computed recursively. In this setting, every node may include a set of neighbourhoods that satisfy the

Algorithm 7 Algorithm Assignment for Extended Laminar Families

Data: rounded instance H, the root node r for a tree in $\mathcal{F}$.

Result: returns either a $(1+\frac{3}{k})$-assignment for all machines in the rounded instance, or report failure.

let $\boldsymbol{\nu}_{in_r}$ denote the input vector for the neighbourhoods of all the machines in r

$U(r, \boldsymbol{\nu}_{in_r}) \leftarrow 1$

for *every machine* w *of* r **do**

  let $\boldsymbol{\nu}_{in_w}$ be the input vector for the neighbourhoods of all the machines in w

  recursively run Algorithm Assignment(H, w) and update T

  aggregate(U, T, r)

end

for *every vector* $\boldsymbol{\nu}$ *in* $\mathcal{V}$ *such that* $U(r, \boldsymbol{\nu}) == 1$ **do**

  run Algorithm 4 on the set of machines in r and update T

end

if *parent*(r) $==$ *NULL* $\wedge\ T(t, \mathbf{0}) == 0$ **then**

  return NULL

end

return T

enclosure property. Instead of just looking at every vector in $\mathcal{V}$ for every marked input vector in U, we now make a call to Algorithm 4 which solves the problem on intervals respecting the enclosure property. The function aggregate is similar to Algorithm 6.

Theorem 4. *Let H be an instance of the problem before rounding with n jobs and m machines. Then, for $k \geqslant 4$ there exists a $(1+\frac{4}{k}+\frac{3}{k^2})$-approximation algorithm for the Min-Max allocation problem when the neighbourhoods of the machines form an extended laminar family. The running time is $\mathcal{O}\left(mn^{2(C+1)} \cdot (1+mn^{C+1}+n^{C+2})\right)$ in which $C \leqslant k^{1.4}$.*

Proof. The correctness of the algorithm follows from the correctness of the two algorithms it is based on, namely Algorithm 4 and Algorithm 5. By the definition of extended laminar families, the allocation of jobs to children of a machine M_i does not affect the other machines in the batch of M_i. Therefore, we can first allocate jobs to the children via a depth first search as in the case of laminar families. When have dealt with all the children of a batch of machines, we proceed as in the case of bipartite permutation graphs. The running time is a mixture of the two time complexities. For every node of the forest, Algorithm Assignment and the function aggregate are called once each. The time complexity of aggregate is as before, but the most significant term in the running time of the Algorithm Assignment has changed. For every vector $\boldsymbol{\nu} \in \mathcal{V}$, Algorithm 4 may be called. Therefore, the running time of the second **for** loop of Algorithm 7 is $\mathcal{O}\left(n^{C+1} \cdot (m+n)n^{2(C+1)}\right) = \mathcal{O}\left((m+n)n^{3(C+1)}\right)$. The total time complexity of the algorithm is $\mathcal{O}\left(m \cdot (n^{2(C+1)} + (m+n)n^{3(C+1)})\right) = \mathcal{O}\left(mn^{2(C+1)} \cdot (1+mn^{C+1}+n^{C+2})\right)$. Finally, the approximation ratio is the maximum of that of laminar families and the bipartite permutation graphs, which is $1+\frac{4}{k}+\frac{3}{k^2}$. □

Remark 9. *The extended laminar family of sets defines a large spectrum of instances with laminar families on one end and the bipartite permutation graphs on the other. In fact, the most extreme case of laminar families is the nested neighbourhoods of machines with a simple chain as its forest representation. We defined this family to better understand the complexity of the scheduling problem. Eventually, we seek a dichotomy for the "easy" instances of the problem. In Section 4.5, we will discuss the idea of a dichotomy briefly.*

4.5 Conclusion and Future Work

In all instances of the problem considered in this chapter, a proper ordering has played an important role. We conjecture that a dynamic programming algorithm similar to the one used in this research provides a PTAS for the case of margined-inclusive intervals. However, we do not know a dichotomy classification for the instances of the problem that admit a PTAS. We ask for a dichotomy of the following form. If H belongs to class X of bipartite graphs, then there is a PTAS for Max-Min allocation problem. Otherwise, there is no PTAS. Towards finding that dichotomy, we suggest at other ordered instance of the resource allocation problems. Besides settling the case for the convex graphs, we propose studying trivially perfect bigraphs and circular arc graphs.

Bibliography

[1] A. A. Ageev and M. Sviridenko. Pipage rounding: A new method of constructing algorithms with proven performance guarantee. *J. Comb. Optim.*, 8(3):307–328, 2004.

[2] N. Alon, Y. Azar, G. J. Woeginger, and T. Yadid. Approximation schemese for schedulling on parallel machines. *Journal of Scheduling*, 1:55–66, 1998.

[3] C. Annamalai, C. Kalaitzis, and O. Svensson. Combinatorial algorithm for restricted max-min fair allocation. In *SODA*, 2015.

[4] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, 2007.

[5] A. Asadpour, U. Feige, and A. Saberi. Santa claus meets hypergraph matchings. In *APPROX-RANDOM*, pages 10–20, 2008.

[6] A. Asadpour and A. Saberi. An approximation algorithm for max-min fair allocation of indivisible goods. In *STOC*, pages 114–121, 2007.

[7] Y. Azar and L. Esptein. On-line machine covering. *Journal of Scheduling*, 1(2):67–77, 1998.

[8] E. Balas. The prize collecting traveling salesman problem. *Networks*, 19(4):621–636, 1989.

[9] E. Balas. The prize collecting traveling salesman problem ii: Polyhedral results. *Networks*, 25(4):199–216, 1995.

[10] N. Bansal and M. Sviridenko. The santa claus problem. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006*, pages 31–40, 2006.

[11] M. Bateni, M. Charikar, and V. Guruswami. Maxmin allocation via degree lower-bounded arborescences. In *STOC*, pages 543–552, 2009.

[12] J.-F. Bérubé, M. Gendreau, and J.-Y. Potvin. A branch-and-cut algorithm for the undirected prize collecting traveling salesman problem. *Networks*, 54(1):56–67, 2009.

[13] I. Bezáková and V. Dani. Allocating indivisible goods. *SIGecom Exchanges*, 5(3):11–18, 2005.

[14] L. Blumrosen and N. Nisan. On the computational power of iterative auctions. In *EC*, pages 29–43, 2005.

[15] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379, 1976.

[16] S. J. Brams and A. D. Taylor. *Fair division - from Cake-cutting to Dispute Resolution.* Cambridge University Press, 1996.

[17] A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph classes: a survey*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 1999.

[18] G. Călinescu, C. Chekuri, M. Pál, and J. Vondrák. Maximizing a submodular set function subject to a matroid constraint (extended abstract). In *IPCO*, pages 182–196, 2007.

[19] P. Cappanera, L. Gouveia, and M. G. Scutellà. The skill vehicle routing problem. In *5th International Conference on Network Optimization, 2011*, number LNCS 6701, pages 354–364. Springer-Verlag, 2011.

[20] A. Chabrier. Vehicle routing problem with elementary shortest path based column generation. *Comput. Oper. Res.*, 33(10):2972–2990, Oct. 2006.

[21] D. Chakrabarty, J. Chuzhoy, and S. Khanna. On allocating goods to maximize fairness. In *FOCS*, pages 107–116, 2009.

[22] D. Chakrabarty and G. Goel. On the approximability of budgeted allocations and improved lower bounds for submodular welfare maximization and gap. In *FOCS*, pages 687–696, 2008.

[23] A. A. Chaves and L. A. N. Lorena. Hybrid metaheuristic for the prize collecting travelling salesman problem. In *Proceedings of the EvoCOP 2008*, number LNCS 4972, pages 123–134. Springer-Verlag, 2008.

[24] L. Chen, K. Jansen, and G. Zhang. On the optimality of approximation schemes for the classical scheduling problem. In *SODA*, pages 657–668, 2014.

[25] E. Choi and D.-W. Tcha. A column generation approach to the heterogeneous fleet vehicle routing problem. *Computers & Operations Research*, 34(7):2080 – 2095, 2007.

[26] W. Cook and J. L. Rich. A parallel cutting-plane algorithm for the vehicle routing problem with time windows. Technical report, 1999.

[27] P. Cramton, Y. Shoham, and R. Steinberg, editors. *Combinatorial Auctions*. MIT Press, forthcoming 2005.

[28] H. Crowder and M. W. Padberg. Solving large-scale symmetric travelling salesman problems to optimality. *Management Science*, 26:495–509, 1980.

[29] J. Csirik, H. Kellerer, and G. Woeginger. The exact lpt-bound for maximizing the minimum completion time. *Operations Research Letters*, 11(5):281–287, 1992.

[30] G. Desaulniers, J. Desrosiers, and M. M. Solomon, editors. *Column generation*. GERAD 25th anniversary series. Springer, New York, 2005. GERAD : Groupe d'études et de recherche en analyse des décisions.

[31] M. Desrochers, J. Desrosiers, and M. Solomon. A new optimization algorithm for the vehicle routing problem with time windows. *Oper. Res.*, 40(2):342–354, Mar. 1992.

[32] B. Deuermeyer, D. K. Friesen, and M. A. Langston. Scheduling to maximize the minimum processor finish time in a multiprocessor system. *SIAM Journal of Algebraic Discrete Methods*, 3(2):190–196, 2 1982.

[33] S. Dobzinski, N. Nisan, and M. Schapira. Approximation algorithms for combinatorial auctions with complement-free bidders. *Math. Oper. Res.*, 35(1):1–13, 2010.

[34] S. Dobzinski and M. Schapira. An improved approximation algorithm for combinatorial auctions with submodular bidders. In *SODA*, pages 1064–1073, 2006.

[35] T. Ebenlendr, M. Krcál, and J. Sgall. Graph balancing: a special case of scheduling unrelated parallel machines. In *SODA*, pages 483–490, 2008.

[36] J. Edmonds. Path, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[37] U. Feige. On allocations that maximize fairness. In *SODA*, pages 287–293, 2008.

[38] U. Feige. On estimation algorithms vs approximation algorithms. In *FSTTCS*, pages 357–363, 2008.

[39] U. Feige. On maximizing welfare when utility functions are subadditive. *SIAM J. Comput.*, 39(1):122–142, 2009.

[40] U. Feige and J. Vondrák. Approximation algorithms for allocation problems: Improving the factor of 1 - 1/e. In *FOCS*, pages 667–676, 2006.

[41] D. Feillet, P. Dejax, M. Gendreau, and C. Gueguen. An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems. *Networks*, 44(3):216–229, 2004.

[42] M. Fischetti, J. J. S. González, and P. Toth. Solving the orienteering problem through branch-and-cut. *INFORMS J. on Computing*, 10(2):133–148, May 1998.

[43] M. Fischetti, J. J. Salazar, and P. Toth. A branch-and-cut algorithm for the symmetric generalized travelling salesman problem. *Operations Research*, 45(3):378–393, 1997.

[44] M. Fischetti and P. Toth. An additive approach for the optimal solution of the prize-collecting travelling salesman problem, 1988.

[45] L. Fleischer, M. X. Goemans, V. S. Mirrokni, and M. Sviridenko. Tight approximation algorithms for maximum general assignment problems. In *SODA*, pages 611–620, 2006.

[46] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, January 1979.

[47] E. Q. Geir Hasle, Knut-Andreas Lie, editor. *Geometric Modelling, Numerical Simulation, and Optimization: Applied Mathematics at SINTEF*. Springer-Verlag Berlin Heidelberg, 2007.

[48] M. Gendreau, G. Laporte, and F. Semet. A branch-and-cut algorithm for the undirected selective traveling salesman problem. *Networks*, 32(4):263–273, 1998.

[49] C. A. Glass and H. Kellerer. Parallel machine scheduling with job assignment restrictions. *Naval Research Logistics*, 54(23):250–257, 2007.

[50] F. Glover. Maximum matching in a convex bipartite graph. *Naval research Logistics Quarterely*, 14:313–316, 1967.

[51] M. X. Goemans. The steiner polytope and related polyhedra. *Mathematical Programming*, 63:157–182, 1994.

[52] B. Golden, S. Raghavan, and E. A. Wasil. *The vehicle routing problem : latest advances and new challenges*. Operations research/Computer science interfaces series, 43. Springer, 2008.

[53] D. Golvin. Max-min fair allocation of indivisible goods. Technical Report CMU-CS-05-144, Carnegie Mellon University, 2005.

[54] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5(2):287–326, 1979.

[55] G. Gutin, P. Hell, A. Rafiey, and A. Yeo. A dichotomy for minimum cost graph homomorphisms. *Eur. J. Comb.*, 29(4):900–911, 2008.

[56] G. Gutin and A. P. Punnen, editors. *The Traveling Salesman Problem and its Variations*. Combinatorial optimization. Kluwer Academic, Dordrecht, London, 2002.

[57] M. Habib, R. M. McConnell, C. Paul, and L. Viennot. Lex-bfs and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theor. Comput. Sci.*, 234(1-2):59–84, 2000.

[58] B. Haeupler, B. Saha, and A. Srinivasan. New constructive aspects of the lovász local lemma. *J. ACM*, 58(6):28, 2011.

[59] P. R. Halmos and H. E. Vaughan. The marriage problem. *American Journal of Mathematics*, pages 214–215, 1950.

[60] K. Halse. *Modeling and solving complex vehicle routing problems*. PhD thesis, Technical University of Denmark, 1992.

[61] P. E. Haxell. A condition for matchability in hypergraphs. *Graphs and Combinatorics*, 11(3):245–248, 1995.

[62] P. Hell, B. Mohar, and A. Rafiey. Ordering without forbidden patterns. In A. S. Schulz and D. Wagner, editors, *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 554–565. Springer, 2014.

[63] S. Hong. *A Linear Programming Approach for the Traveling Salesman Problem*. PhD thesis, John Hopkins University, Baltimore, Maryland, USA, 1972.

[64] E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM*, 23(2):317–327, 1976.

[65] H.-C. Hwang, S. Y. Chang, and K. Lee. Parallel machine scheduling under a grade of service provision. *Computers & OR*, 31(12):2055–2061, 2004.

[66] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001.

[67] S. Irnich and D. Villeneuve. The shortest-path problem with resource constraints and k-cycle elimination for k $\geqslant$ 3. *INFORMS Journal on Computing*, 18(3):391–406, 2006.

[68] K. Jansen. An eptas for scheduling jobs on uniform processors: Using an milp relaxation with a constant number of integral variables. *SIAM J. Discrete Math.*, 24(2):457–485, 2010.

[69] K. Jansen and M. Mastrolilli. Scheduling unrelated parallel machines: linear programming strikes back. Technical Report 1004, University of Kiel, 2010.

[70] K. Jansen and L. Porkolab. Improved approximation schemes for scheduling unrelated parallel machines. *Math. Oper. Res.*, 26(2):324–338, 2001.

[71] K. Khodamoradi and R. Krishnamurti. Prize collecting travelling salesman problem - fast heuristic separations. In *Proceedings of 5th the International Conference on Operations Research and Enterprise Systems (ICORES 2016), Rome, Italy, February 23-25, 2016.*, pages 380–387, 2016.

[72] K. Khodamoradi, R. Krishnamurti, A. Rafiey, and G. Stamoulis. PTAS for ordered instances of resource allocation problems. In A. Seth and N. K. Vishnoi, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, December 12-14, 2013, Guwahati, India*, volume 24 of *LIPIcs*, pages 461–473. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[73] S. Khot, R. J. Lipton, E. Markakis, and A. Mehta. Inapproximability results for combinatorial auctions with submodular utility functions. *Algorithmica*, 52(1):3–18, 2008.

[74] S. Khot and A. K. Ponnuswami. Approximation algorithms for the max-min allocation problem. In *APPROX-RANDOM*, pages 204–217, 2007.

[75] N. Kohl, J. Desrosiers, O. B. G. Madsen, M. M. Solomon, and F. Soumis. 2-path cuts for the vehicle routing problem with time windows. *Transportation Science*, 33(1):101–116, 1999.

[76] N. Kohl and O. B. G. Madsen. An optimization algorithm for the vehicle routing problem with time windows based on lagrangian relaxation. *Oper. Res.*, 45(3):395–406, June 1997.

[77] A. Land. The solution of some 100-city travelling salesman problems. Technical report, London School of Economics, London, UK, 1979.

[78] J. Larsen. *Parallelization of the Vehicle Routing Problem with Time Windows*. PhD thesis, Technical University of Denmark, 1999.

[79] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. *Handbooks in operations research and management science*, 4:445–522, 1993.

[80] K. Lee, J. Y. Leung, and M. L. Pinedo. A note on graph balancing problems with restrictions. *Inf. Process. Lett.*, 110(1):24–29, 2009.

[81] B. Lehmann, D. J. Lehmann, and N. Nisan. Combinatorial auctions with decreasing marginal utilities. In *EC*, pages 18–28, 2001.

[82] H. W. Lenstra. Integer programming with fixed number of variables. *Mathematics of Operations Research*, 8:538–548, 1983.

[83] J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Math. Program.*, 46:259–271, 1990.

[84] C. Lenté, M. Liedloff, A. Soukhal, and V. T'Kindt. On an extension of the sort & search method with application to scheduling theory. *Theor. Comput. Sci.*, 511:13–22, 2013.

[85] R. J. Lipton, E. Markakis, E. Mossel, and A. Saberi. On approximately fair allocations of indivisible goods. In *EC*, pages 125–131, 2004.

[86] M. Mastrolilli and G. Stamoulis. Restricted max-min fair allocations with inclusion-free intervals. In J. Gudmundsson, J. Mestre, and T. Viglas, editors, *Computing and Combinatorics - 18th Annual International Conference, COCOON 2012, Sydney, Australia, August 20-22, 2012. Proceedings*, volume 7434 of *Lecture Notes in Computer Science*, pages 98–108. Springer, 2012.

[87] J. Meidanis, O. Porto, and G. P. Telles. On the consecutive ones property. *Discrete Applied Mathematics*, 88(1-3):325–354, 1998.

[88] G. Muratore, U. M. Schwarz, and G. J. Woeginger. Parallel machine scheduling with nested job assignment restrictions. *Oper. Res. Lett.*, 38(1):47–50, 2010.

[89] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization*. Wiley interscience series in discrete mathematics and optimization. Wiley, 1988.

[90] T. T. Nguyen, M. Roos, and J. Rothe. A survey of approximability and inapproximability results for social welfare optimization in multiagent resource allocation. *Ann. Math. Artif. Intell.*, 68(1-3):65–90, 2013.

[91] T. O'Neil and S. Kerlin. A simple $o(2^{\mathrm{sqrt(x)}})$ algorithm for partition and subset sum. In *FCS*, pages 55–58, 2010.

[92] J. Ou, J. Y.-T. Leung, and C.-L. Li. Scheduling parallel machines with inclusive processing set restrictions. *Naval Research Logistics*, 55(4):328–338, June 2008.

[93] M. W. Padberg and S. Hong. On the symmetric traveling salesman problem: a computational study. *Mathematical Programming Study*, 12:778–107, 1980.

[94] L. Polacek and O. Svensson. Quasi-polynomial local search for restricted max-min fair allocation. In *ICALP (1)*, pages 726–737, 2012.

[95] G. Reinelt. Tsplib: A traveling salesman problem library. *ORSA J. Comput.*, pages 376–384, 1991.

[96] J. Rodrigue, C. Comtois, and B. Slack. *The Geography of Transport Systems*. The Geography of Transport Systems. Routledge, 2009.

[97] L.-M. Rousseau, M. Gendreau, G. Pesant, and F. Focacci. Solving vrptws with constraint programming based column generation. *Annals of Operations Research*, 130(1):199–216, 2004.

[98] B. Saha and A. Srinivasan. A new approximation technique for resource-allocation problems. In *ICS*, pages 342–357, 2010.

[99] U. M. Schwarz. A ptas for scheduling with tree assignment restrictions. *CoRR*, abs/1009.4529, 2010.

[100] J. Sgall. Randomized on-line scheduling of parallel jobs. *J. Algorithms*, 21(1):149–175, 1996.

[101] J. P. Spinrad. *Efficient graph representations*. Fields Institute monographs. American Mathematical Society, Providence, RI, 2003.

[102] O. Svensson. Santa claus schedules jobs on unrelated machines. In *STOC*, pages 617–626, 2011.

[103] E. D. Taillard. A heuristic column generation method for the heterogeneous fleet vrp. *RAIRO - Operations Research*, 33(1):1–14, 001 1999.

[104] Z. Tan, Y. He, and L. Epstein. Optimal on-line algorithms for the uniform machine scheduling problem with ordinal data. *Inf. Comput.*, 196(1):57–70, 2005.

[105] J. Verschae and A. Wiese. On the configuration-lp for scheduling on unrelated machines. *J. Scheduling*, 17(4):371–383, 2014.

[106] J. Vondrák. Optimal approximation for the submodular welfare problem in the value oracle model. In *STOC*, pages 67–74, 2008.

[107] G. J. Woeginger. A polynomial-time approximation scheme for maximizing the minimum machine completion time. *Operations Research Letters*, 20(4):149 – 154, 1997.

[108] L. Wolsey. *Integer Programming*. John Wiley & Sons, New York, 1998.

Appendix A

The Code Summary

A.1 Major Classes and Functions

The branch-and-cut algorithm of Chapter 3 has been implemented in C++ using the IBM CPLEX ILOG library. In this appendix, we mention some of the central classes used in the code as well as the main functions in each class. The description is given in the form of a summarized header file of each class alongside C++ comments for explanation. The source code for the software is publicly available at https://github.com/KamyarK/SVRP.

A.1.1 The Solver Class

```
class Solver {
    // This class is the driver of the code. It has functionality for both
    // running the column generation algorithm on an LP relaxation of the
    // Skill Vehicle Routing Problem (SVRP), or run the entire branch-and-cut
    // algorithm for the Prize Collecting Travelling Salesman Problem (PCTSP)
    // for a driver and the set of customers that the driver can service.

public:
        void readFile();
            // This function reads the input file given to the main program
            // as an argument and stored in the private data member 'filename'.

        double MST(vector<int> subset);
            // This function computes a Minimum Spanning Tree (MST) on the
            // set of the nodes in the graph indicated by the vector 'subset'.

        void printSubsets(int subset_num);
            // This function outputs the list of all subsets (columns) currently
            // used in the column generation model. The input variable to the
            // function, 'subset_num', indicated the number of such columns.
            // Also, these columns are stores in the private data member
```

```
            // 'service_subsets'.

        double Optimize();
            // This function runs the column generation algorithm for the
            // instance of the SVRP given in 'filename'. It may instantiate
            // an instance of the class 'PTSP' in order to solve the subproblem
            // and obtain a better column generation bound.

        void PCTSP(int wn);
            // This function instantiates an instance of the class 'PTSP' for
            // a given driver, 'wn', and the set of nodes to whom this driver cab
            // provide service.

        inline void printResults(const time_t &beginInterval, const double
        &objVal, const double &bestLowerBound, const double &Round);
            // This function outputs the best lower and upper bounds found for
            // a node in the column generation algorithm together with the
            // elapsed time.

private:
        char *filename, *logfile;
            // The input file and the log file.

        bool serviceable[MAX_WORKER][MAX_JOB];
            // Shows whether the job is serviceable by the driver.

        vector< vector<int> > service_subsets;
            // Stores the current columns.

        inline void checkExpand(int subset_num);
            // This function doubles the size of the columns if the capacity is
            // reached. Note that the physical address of the data changes in
            // the processed.

};
```

A.1.2 The GraphUtil Class

```
class GraphUtil {
    // This class provides some utility functions for processing the graphs
    // that the software encounters in solving SVRP or PCTSP.

public:
        std::list<std::list<int> > components();
            // This function returns the connected components of the instance
            // of the 'GraphUtil' class represented by '*this'.
```

```
        std::list<std::list<int> > shrink(const bool verbose);
            // This function implements the shrinking heuristic on the instance
            // and returns the components after the graph is shrunk. The Boolean
            // variable 'verbose' indicates to the function whether or not it
            // should produce messages along the way for debugging purposes.

        double GW_PTSP(std::vector<int> &tree, double *treeMat);
            // This functions implements the algorithm by Hedge, Indyk, and
            // Schmidt for the Prize Collecting Steiner Tree Problem (PCSTP)
            // using Goemans and Williamson scheme. This method is a heuristic
            // approach to solving the PCSTP.

        double LS_PTSP(std::vector<int> &cycle, double *cycleMat);
            // This function implements a local search heuristic for the PCTSP.
            // It makes calls to some utility functions, namely 'forceOne',
            // 'oneOpt', 'twoOpt', and 'oneSwap'.

        void printComp(std::list< std::list<int> > l);
            // This function outputs the connected components represented by
            // the input variable 'l'.

        int num_component(std::list< std::list<int> > l);
            // This function counts the number of active components of the graph
            // represented by the input variable 'l'. By active, we mean those
            // components that are not single nodes and do not contain the depot.
    private:
        void StrongPrune(double *treeMat, double *NW, bool *visited, int v);
            // This function implements the strong pruning required by the
            // function 'GW_PTSP'.

        void removeTree(double *treeMat, int r);
            // This function implements the removal phase of 'GW_PTSP'.

        double shortcut(double *treeMat, bool *visited, int r);
            // This function finds the shortcuts in a given tour and modifies
            // it to avoid repeated nodes.

        bool forceOne(std::vector<int> &subset_mat, double *cycleMat,
        std::list<int> &cycle, double &tourCost);
            // This function forces the tour to pick at least one node.

        double oneOpt(std::vector<int> &subset_mat, double *cycleMat,
        std::list<int> &cycle, double &tourCost);
            // This function adds one most beneficial node to a given tour.

        double twoOpt(std::vector<int> &subset_mat, double *cycleMat,
```

```
        std::list<int> &cycle, double &tourCost);
            // This function adds two nodes to a given tour and removes one in
            // a greedy manner to maximize the net benefit.

        double oneSwap(std::vector<int> &subset_mat, double *cycleMat,
        std::list<int> &cycle, double &tourCost);
            // This function swaps an existing node of the given tour with an
            // uncovered node in a greedy manner to maximize the net benefit.

};
```

A.1.3 The ISubProblem Class

```
class ISubProblem {
    // This class is  abstract base class for the PCTSP, the subproblem of the
    // column generation method. This class dictated to all sub-classes to
    // implement the member function 'Optimize'.

public:
        virtual double Optimize(vector<int> &subset_mat, double& tourCost, double
        &interval, const OPT_TYPE &optType, const Params params) = 0;
            // This function is purely virtual. It specifies the layout of any
            // 'Optimize' function to be implemented in the sub-classes.

protected:
        int worker_no;
            // The driver ID .

        int num_worker;
            // The number of the  drivers.
        int num_job;
            // The number of the clients.

        bool *serviceable;
            // Shows whether the job is serviceable by the driver.

        double *cost;
            // Stores the distance c_{ij} between the nodes i and j.

        double *benefit;
            // Stores the prizes or beta values for nodes.
};
```

A.1.4 The PTSP Class

```
class PTSP: public ISubProblem {
    // This class inherits from the abstract base class 'ISubProblem' and
    // implements the optimize function that can be used in different modes:
    //    1) to use the branch-and-cut to solve the PCTSP,
    //    2) to solve an LP relaxation of a node of the branch-and-cut tree,
    //    3) to use CPLEX ILP solver to solve PCTSP,
    //    4) to solve PCTSP using a local search heuristic algorithm.

public:
        double Optimize(vector<int> &subset, double &tourCost, double &interval,
        const OPT_TYPE &optType, const Params params);
            // This function implements the 'Optimize' function required by the
            // super-class ISubProblem. Based on the value of 'OPT_TYPE', one of
            // the four solutions mentioned above are attempted.

private:
        double Optimize_Branch_and_Cut(vector<int> &subset, double &tourCost,
        double &interval, const Params params);
            // This function uses the branch-and-cut algorithm to solve PCTSP
            // to optimality. The tour cost and the time spent in this portion
            // of the code are returned in 'tourCost' and 'interval'
            // respectively.

        double Optimize_LP(vector<int> &subset, double &tourCost, double
        &interval, const Params params);
            // This function uses the CPLEX to solve the LP relaxation of PCTSP.
            // The tour cost and the time spent in this portion of the code are
            // returned in 'tourCost' and 'interval' respectively.

        double Optimize_ILP(vector<int> &subset, double &tourCost, double
        &interval);
            // This function uses the CPLEX ILOG ILP solver to solve PCTSP
            // to optimality. The tour cost and the time spent in this portion
            // of the code are returned in 'tourCost' and 'interval'
            // respectively.

        double Heuristic(vector<int> &subset, double &tourCost, double
        &interval);
            // This function uses a local search  heuristic algorithm to solve
            // PCTSP. The tour cost and the time spent in this portion of the
            // code are returned in 'tourCost' and 'interval' respectively.

        int Separate_SEC(IloModel &model, IloNumVarArray &x, IloNumVarArray &y,
        IloNumArray &xVals, IloNumArray &yVals, IloRangeArray &con, const
        vector<int> &realIndex, int num_servable, int &poolSize, IloRangeArray
        &poolOfConstraints, Branch_and_CutNode &P);
            // This function takes an integral solution returned by the CPLEX
```

```
    // model and uses connected components algorithms to find violated
    // Subtour Elimination Constraints (SECs). For memory management
    // reasons in the branch-and-cut algorithm, all the constraints
    // are added to a pool of constraints instead of the CPLEX model.

int SEC(IloModel &model, IloNumVarArray &x, IloNumVarArray &y, const
vector<int> &realIndex, double *edgeMat, int num_servable);
    // Similar to the above function, but used outside the branch-and-cut
    // algorithm, e.g. in the LP relaxation of the PCTSP.

int Separate_LP_GSEC(IloModel &model, IloNumVarArray &x, IloNumVarArray
&y, IloNumArray &xVals, IloNumArray &yVals, IloRangeArray &con, const
vector<int> &realIndex, int num_servable, int &poolSize, IloRangeArray
&poolOfConstraints, Branch_and_CutNode &P);
    // This function takes a fractional solution and solves some LP
    // problems to find violated Generalized Subtour Elimination
    // Constraints (GSECs). For memory management
    // reasons in the branch-and-cut algorithm, all the constraints
    // are added to a pool of constraints instead of the CPLEX model.

int Separate_LP_GSEC(IloModel &model, IloNumVarArray &x, IloNumVarArray
&y, IloNumArray &xVals, IloNumArray &yVals, const vector<int> &realIndex,
int num_servable);
    // Similar to the above function, but used outside the
    // branch-and-cut.

int GSEC(IloModel &model, IloNumArray &xVals, IloNumArray &yVals, const
vector<int> &realIndex, int num_edges, int num_servable, list< list<int>
> &l);
    // This function calls the appropriate LP separation function based
    // on the method of optimization, receives back the list of violated
    // GSECs, and adds them to the CPLEX model.

int Separate_heuristic_GSEC(IloModel &model, IloNumVarArray &x,
IloNumVarArray &y, IloNumArray &xVals, IloNumArray &yVals, IloRangeArray
&con, const vector<int> &realIndex, int num_servable, bool tempVerbose,
int &poolSize, IloRangeArray &poolOfConstraints, Branch_and_CutNode &P);
    // This function takes a fractional solution and uses the shrinking
    // heuristic algorithm to find violated GSECs. For memory management
    // reasons in the branch-and-cut algorithm, all the constraints
    // are added to a pool of constraints instead of the CPLEX model.

int Separate_heuristic_GSEC(IloModel &model, IloNumVarArray &x,
IloNumVarArray &y, IloNumArray &xVals, IloNumArray &yVals, const
vector<int> &realIndex, int num_servable, bool tempVerbose);
    // Similar to the above function, but used outside the
    // branch-and-cut.
```

```
int GSEC_heuristic(const IloNumArray &xVals, const IloNumArray &yVals,
const vector<int> &realIndex, int num_servable, list< list<int> > &l,
const bool verbose);
    // This function calls the appropriate heuristic separation function
    // based on the method of optimization, receives back the list of
    // violated GSECs, and adds them to the CPLEX model.

int Separate_heuristic_Blossoms(IloModel &model, IloNumVarArray &x,
IloNumVarArray &y, IloNumArray &xVals, IloNumArray &yVals, IloRangeArray
&con, const vector<int> &realIndex, int num_servable, int &poolSize,
IloRangeArray &poolOfConstraints, Branch_and_CutNode &p);
    // This function takes a fractional solution and uses the
    // odd-component heuristic algorithm to find violated Primitive Comb
    // Inequalities. For memory management reasons in the branch-and-cut
    // algorithm, all the constraints are added to a pool of constraints
    // instead of the CPLEX model.

int Separate_heuristic_Blossoms(IloModel &model, IloNumVarArray &x,
IloNumVarArray &y, IloNumArray &xVals, IloNumArray &yVals, const
vector<int> &realIndex, int num_servable);
    // Similar to the above function, but used outside the
    // branch-and-cut.

int Blossom_heuristic(const IloNumArray &xVals, const IloNumArray &yVals,
const vector<int> &realIndex, int num_servable, list<blossom> &l);
    // This function calls the appropriate heuristic separation function
    // based on the method of optimization, receives back the list of
    // violated Primitive Comb Inequalities, and adds them to the CPLEX
    // model.

void branchOnX(IloModel &model, IloNumVarArray &x, IloNumArray &xVals,
IloRangeArray &con, const vector<int> &realIndex, int &problemCount, int
num_servable, int &poolSize, IloRangeArray &poolOfConstraints,
Branch_and_CutNode &P, list<Branch_and_CutNode> &Problems);
    // This function receives a fractional solution and makes a call to
    // the function 'chooseX'. After receiving the 'x' chosen variable,
    // it creates two subproblems, one for 'x = 0' and the other for
    // 'x = 1'. Then, it adds the problems to the queue of remaining
    // problems to be solved, based on the mode of traversal.

void branchOnY(IloModel &model, IloNumVarArray &y, IloNumArray &yVals,
IloRangeArray &con, const vector<int> &realIndex, int &problemCount,
int &poolSize, IloRangeArray &poolOfConstraints, Branch_and_CutNode &P,
list<Branch_and_CutNode> &Problems);
    // This function receives a fractional solution and makes a call to
    // the function 'chooseY'. After receiving the 'y' chosen variable,
```

```
    // it creates two subproblems, one for 'y = 0' and the other for
    // 'y = 1'. Then, it adds the problems to the queue of remaining
    // problems to be solved, based on the mode of traversal.

int chooseX(IloNumArray &xVals, int &num_servable, const vector<int>
&realIndex, Branch_and_CutNode &P);
    // This function chooses an 'x' variable based on the model of
    // branching and returns it.

int chooseY(IloNumArray &yVals, const vector<int> &realIndex,
Branch_and_CutNode &P);
     // This function chooses an 'y' variable based on the model of
    // branching and returns it.

int randomBranchOnX(IloNumArray &xVals, int &num_servable, const
vector<int> &realIndex, Branch_and_CutNode &P);
    // This function implements the 'Random Branch' mode of branching
    // on 'x' variables, choosing any random 'x'.

int randomBranchOnY(IloNumArray &yVals, const vector<int> &realIndex,
Branch_and_CutNode &P);
    // This function implements the 'Random Branch' mode of branching
    // on 'y' variables, choosing any random 'y'.

int greedyBranchOnX(IloNumArray &xVals, int &num_servable, const
vector<int> &realIndex, Branch_and_CutNode &P);
    // This function implements the 'Greedy Branch' mode of branching
    // on 'x' variables, choosing any 'x' with a value closest to 0.5.

int greedyBranchOnY(IloNumArray &yVals, const vector<int> &realIndex,
Branch_and_CutNode &P);
    // This function implements the 'Greedy Branch' mode of branching
    // on 'y' variables, choosing any 'y' with a value closest to 0.5.

int smartBranchOnX(IloNumArray &xVals, int &num_servable, const
vector<int> &realIndex, Branch_and_CutNode &P);
    // This function implements the 'Smart Branch' mode of branching
    // on 'x' variables. This is due to Gendreau et al.

int smartBranchOnY(IloNumArray &yVals, const vector<int> &realIndex,
Branch_and_CutNode &P);
    // This function implements the 'Smart Branch' mode of branching
    // on 'y' variables. This is due to Gendreau et al.

void makeGraph(double *edgeMat, double *vertices);
    // This function generates an empty graph based on '*this'. This is
    // done by filling the elements of the edge matrix '*edgeMat' and
```

```
        // the vertices in '*vertices'.

    void makeGraph(double *edgeMat, double *vertices, IloNumArray &yVals,
    const vector<int> &realIndex);
        // This function generates graph based on '*this' and 'y' values.
        // This is done by filling the elements of the edge matrix '*edgeMat'
        // and the vertices in '*vertices'.

    inline bool isSolutionIntegral(const IloNumArray &xVals, const
    IloNumArray &yVals, const vector<int> &realIndex, int num_servable);
        // This function returns true if both 'x' and 'y' values are
        // integral.

    inline bool isXIntegral(const IloNumArray &xVals, const vector<int>
    &realIndex, int num_servable);
        // This function returns true if 'x' values are integral.

    inline bool isYIntegral(const IloNumArray &yVals, const vector<int>
    &realIndex);
        // This function returns true if 'y' values are integral.

    inline void updateConstraints (IloRangeArray &con, const IloRangeArray
    &poolOfConstraints, Branch_and_CutNode &P);
        // This function updates the list of constraints of a node in the
        // branch-and-cut tree after new violated cuts are detected and
        // introduced in the pool of constraints.
};
```