Selected Applications of Integer Programming: A Computational Study

Geselecteerde toepassingen van geheeltallig programmeren:
Een computationele studie

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van
doctor aan de Universiteit Utrecht
op gezag van de Rector Magnificus, Prof. Dr. H.O. Voorma,
ingevolge het besluit van het College voor Promoties
in het openbaar te verdedigen
op maandag 11 september 2000 des ochtends te 10.30 uur

door

Abraham Michiel Verweij

geboren op 29 oktober 1971
te 's-Gravenhage

promotor: Prof. Dr. J. van Leeuwen, Faculteit Wiskunde & Informatica
co-promotoren: Dr. Ir. K.I. Aardal, Faculteit Wiskunde & Informatica
Dr. G. Kant, Ortec Consultants bv, Gouda

ISBN 90-393-2452-2

Contents

Preface

We consider solving instances of mathematical optimisation problems by computer. This involves making a mathematical model for the problem at hand, designing an algorithm that is based on this model, and implementing and testing the algorithm. Often this leads to new ideas concerning the model, or the algorithms we chose to use, or both, and then we have to adjust our solutions to the newly gained insights. Clearly, such a study cannot succeed without the availibility of interesting mathematical optimisation problems.

I consider myself to be a lucky PhD student, because I never had to invent my own problems. My PhD-topic fitted in the European ALCOM-IT project (ALCOM-IT stands for ALgorithms and COMplexity in Information Technology), which had as a main goal to develop algorithms for industrial applications. My co-promoter Goos Kant participated in this project on behalf of Ortec Consultants bv, Gouda, The Netherlands. He ensured that I always had some problems on which people from Ortec were working as well.

The first problem Goos brought to my attention originated from an anonymous European airplane manufacturer that has production and assembly sites all over Europe. While developing algorithms for this problem I first learned about integer programming and branch-and-bound algorithms. Using instances of this problem I solved my first real linear programs. Although I learned a lot while working on this problem, it is not treated in this thesis. It is described in a technical report [118].

The second problem Goos brought into my life was an "easy" problem. Goos used to say that instances of it could be specified by "just two matrices of input data". We named this problem "The Van Gend & Loos Problem", after the company where it emerged. Chapters 6 and 7 treat this problem in detail. It turns out that there is more to it than the two matrices Goos was referring to, which described the distances and the demands between customers. Travel times between customers and a fleet of vehicles that are stationed at a subset of the sites of the customers complete the description of the problem. We will show that this problem is $\mathcal{NP}$-hard, and that even finding a feasible solution to it is $\mathcal{NP}$-hard. And we will fail miserably in our effort to find provably optimal solutions. Having said this, we are able to compute satisfactory solutions to instances that are roughly half the size of the one given by Goos using pretty sophisticated algorithmic techniques.

After having studied this problem for a while I decided to write my own

code for it from scratch (except for the LP solver, that is). I felt that I did not have full freedom in algorithmic design when using the frameworks that were available. Apart from this I always underestimate the amount of work that goes into implementing algorithms, and overestimate my own coding abilities. This was in April 1998. Looking back, there is one big advantage of having your own code, namely, that you really know what goes on after you hit return on the keyboard to set the computer to work (and work, and work). In some way it also increases the fun of inspecting the result of the algorithm.

In order to develop and test my own code I needed instances of a problem with a relatively clean problem structure. At first I used the knapsack problem for this purpose, but after some time I decided that I needed a problem with a graph structure, and started to use instances of a map labelling problem. Apart from being interesting problems in their own right, instances of this map labelling problem were easily accessible to me as Steven van Dijk, one of my PhD colleagues in the Computer Science Department, was developing genetic algorithms for solving them. A variant of the map labelling problem can be reformulated as an independent set problem. It were these derived independent set instances that I used for testing my code. Even the earlier versions of it were able to solve moderately sized map labelling instances within reasonable time. Chapter 5 treats our approach to the map labelling problem in detail. At that time, I did not know that independent set problems are notoriously hard to solve to optimality. Fortunately, my co-promoter Karen Aardal did know this.

Developing my own code for integer programming also meant learning about how people have solved integer programming problems in the past, even at times back to before I was born. For me, this truly gave meaning to the word "student" in "PhD-student". I hope that this thesis is as interesting for you to read as it was for me to learn, develop, code, and test all the algorithms that form its contents.

Utrecht, May 2000

Bram Verweij

Acknowledgements

This text has benefited a lot from comments and remarks by Karen Aardal and Jan van Leeuwen on the first draft. Comments by Erik Balder, Stan van Hoesel, Goos Kant, Alexander Martin and Laurence Wolsey further improved the quality of the text.

I thank Karen, first for being an excellent supervisor, and second for sending me to conferences and workshops all over the world during my time as a PhD-student. These conferences and workshops turned out to be very inspiring to me (especially the Aussois workshops), and have played a key role in my choice of algorithmic techniques. I thank Goos for providing me with interesting problems to work on.

I am grateful to my colleagues of the Computer Science Department of Utrecht University for making it such a pleasant place to work as a PhD-student. Special thanks go to Han, Job, João, Michiel, Paul, Robert-Paul, Tycho, Valérie and Jasper Aukes. Finally, I thank Marjan for all the love and support that she gave me during the past four years.

Just before my summer break in 1997 I was moved from a room on the first floor that I shared with my colleagues Robert-Paul and João to a room on the third floor that was all for myself. Afterwards I was sad about having left the first floor, which I thought of as PhD-paradise. In this mood I made a romantic picture of a palm beach on some paradise island on the white board (that is, as far as my artistic capabilities allowed). When I came back from my summer break, the picture was still there and I decided to leave it. My friends spontaneously started to add features to it, and at some point in time I started kidding that I would make it the cover of my thesis, which eventually I did.

The people who contributed to the cover are, as far as I can remember, Karen Aardal, Abdu Basher, Robert-Paul Berretty, Valérie Boor, Babette de Fluiter, Joachim Gudmundsson, Michiel Hagedoorn, Mikael Hammer, Paul Harrenstein, Eveline Helsper, Lennart Herlaar, Stan van Hoesel, Goos Kant, Judith Keijsper, Elisabeth Lenstra, Jan Karel Lenstra, Stefano Leonardi, Han-Wen Nienhuys, Joana Passos, Hans Philippi, João Saraiva, Petra Schuurman, Daniëlle Sent, Job Smeltink, Tycho Strijk, Edwin Veenendaal, Jacques Verriet, Tjark Vredeveld, Marjan de Vries, Chantal Wentink and Wieger Wesselink.

Chapter 1

Introduction

1.1 Problems and Goals

Consider the problem of putting city names on a map. Each name has to be adjacent to the city it belongs to, and no two names are allowed to overlap. The problem is to put as many names on the map as possible, subject to these conditions. This problem is known as a *map labelling problem*.

Next, consider the problem of a merchant that drives around in a van and makes money by buying commodities where they are cheap and selling them in other places where he can make a profit. Assuming that we know the prices for all of the commodities in all of the places and the cost of driving from one place to another, the problem the merchant faces each day is to select a subset of the cities that he can visit in a day, and that maximises the profit he makes. We call this problem the *merchant subtour problem*.

Finally, consider a vehicle routing problem in which we have at our disposal a fleet of trucks that are stationed at several depots, and in which we are given a set of customers with an integral demand for different commodities. Each commodity originates from one customer and is destined for another customer. Goods of each commodity may be split into integral amounts for transportation. Assuming that we know the cost of driving from one place to another, the problem is to transport all demand using the available trucks, minimising the total cost of driving. As this problem models a problem that arises at the Van Gend & Loos company in the Netherlands (and it is the only problem in this thesis that does so) we call it the *Van Gend & Loos problem*.

The problems described above have in common that they can be described as optimisation problems in integral variables subject to linear constraints, with a linear objective function (for details, see Part II). Problems of this kind are known as *integer linear programming problems*. Specific integer linear programming problems have been studied by many authors in the past. For a number of these problems algorithms have been reported that solve them successfully.

Our first goal is to develop algorithms for the specific problems named above.

This includes developing heuristics that only report feasible solutions, and optimisation algorithms that solve them to optimality. Our second goal is to study the behaviour of the algorithms we develop. This involves analysing the time complexity and the quality of the solutions. For the problems we consider it is not clear how to develop algorithms that allow for both a satisfactory analysis of the time complexity and a satisfactory analysis of the quality of the solutions. Indeed, heuristic algorithms often have a nice worst-case time complexity but lack a good guaranteed bound on the solution quality, whereas optimisation algorithms do produce a solution that is as good as is possible but they often do not have a nice worst-case bound on the time complexity, i.e., a bound that guarantees termination within reasonable time. Therefore, in order to measure the behaviour of our algorithms from a perspective different from the worst-case we study them by means of computational experiments.

When designing algorithms for the problems mentioned above we try to take advantage of similar studies of other specific integer linear programming problems that were successful and can be found in the literature. This is our main motivation to study *branch-and-cut* and *branch-and-price* algorithms, as well as *LP rounding*, *local search*, and *tabu search*.

1.2 Outline and Contribution

This thesis is organised as follows. In Part I we describe the core of our optimisation algorithms. In Part II we apply the algorithms presented in Part I to the specific problems of our interest.

Part I starts with Chapter 2 that discusses the generic branch-and-bound algorithm that is used for finding provably optimal solutions, reviews relevant issues of computational complexity, and reviews some results from the theory of linear programming that we rely on. In Chapter 3 we discuss how the branch-and-bound algorithm can be refined for integer linear programming problems, leading to branch-and-cut, branch-and-price, and branch-price-and-cut algorithms. In Chapter 4 we study the *maximum independent set problem*, a classical problem that can be formulated as an integer linear programming problem, and discuss branch-and-cut and LP rounding algorithms for it. We are interested in the maximum independent set problem because it is possible to reformulate the map labelling problem into it.

We proceed in Part II with Chapter 5 where we present our algorithms for the map labelling problem. These algorithms are a refinement of the algorithms from Chapter 4 for the maximum independent set problem. In Chapter 6 we present a tabu search heuristic and a branch-price-and-cut algorithm for the merchant subtour problem. Finally, a branch-and-price algorithm, and a related branch-and-price based heuristic for the Van Gend & Loos problem are presented in Chapter 7. The algorithms from Chapter 7 use the ones from Chapter 6 as subroutines.

The models and algorithms that are discussed in Part I are well-documented in the literature. However, when implementing them one has to make numer-

ous choices concerning details. Sometimes this has resulted in original features of our code. In particular, we like to mention the scheme for re-computing variable bounds in each node of the branch-and-bound algorithm after an improving primal solution has been found, treated in Section 3.2.2, the improved reduced cost arguments for tightening variable bounds, treated in Section 3.2.3 (these were developed together with Strijk [107]), the combined variable/GUB branching scheme from Section 3.2.5, the minimum regret rounding heuristic for maximum independent sets, treated in Section 4.2.2, the scheme for identifying an odd hole and using a path-decomposition for lifting odd hole inequalities, treated in Section 4.3.3. To the best of our knowledge, these ideas have not been reported on before.

Our contribution in Part II is the following. For the map labelling problem, we are the first to report on an optimisation algorithm that can solve instances of the map labelling problem with up to 950 cities on a standard map to optimality within reasonable time. An earlier version of our algorithm for map labelling was reported on in Verweij and Aardal [117]. The recursive technique for setting variables from Section 5.2.3 is new. The merchant subtour problem and the Van Gend & Loos problem are original, and all models and algorithms for it presented in Chapters 6–7 are of our own design.

A major part of the work behind this thesis has been spent on implementing and testing all the algorithms to obtain experimental data. This effort is accumulated in the sections on computational results. We ask the reader to pay special attention to all the tables and figures in this thesis that describe the behaviour of the algorithms we present, as only those tables and figures give insight into the practical value of the algorithms to which they refer. Finally, we mention that we are the first to report on computational experiments using the separation algorithm for finding maximally violated mod-k cuts of Caprara, Fischetti, and Letchford discussed in Section 3.3.5. For this reason we will study the yield of this particular separation algorithm with a bit more detail.

1.3 Computational Environment

As this is a computational study, some words about the system on which we performed our computations are in order. All our computations were executed on Sun workstations, using C++ as a programming language (see Stroustrup [109]). Unless otherwise mentioned, the Sun workstation was a 360MHz Sun Enterprise 2 with 512 megabyte main memory and eight gigabyte swap space running UNIX as OS. We use the Gnu gcc compiler, version 2.95.2, with maximal compiler optimisations (–O3).

Our implementation consists of approximately 24350 non-blank lines of C++ code in total, spread over 94 modules. Chapters 2 and 3 involve approximately 11300 non-blank lines of C++ code, also including elementary data structures such as linked lists, skip lists [96], heaps, and hash-tables. Chapters 4, 5, 6, and 7 involve approximately 3300, 1600, 6450, and 1700 lines of code, respectively.

When implementing algorithms one has to be aware of the possibility that

there are bugs in the implementation. Indeed, during the development of our implementation plenty of time was spend on tracing and eliminating bugs. Unfortunately, this does not give a guarantee that our implementation is fully bug-free.

We have included in our implementation numerous checks that verify intermediate results. As these checks verify the consistency of what we compute and are mainly inspired by common sense we will refer to them as *sanity checks*. Our sanity checks include, among other things, verifying feasibility of solutions produced by our algorithms, watching bound clashes (i.e., bounds that contradict with feasible solutions produced by heuristics) and verifying the violation of reported violated inequalities (see Section 3.3). Equally important is that we do not turn these sanity checks off to obtain a better performance in terms of absolute CPU time. We can afford this because the CPU time used by the sanity checks does not determine the order of magnitude of the total CPU time. For all the experiments reported on in this thesis, our algorithms terminated normally and no abnormalities were reported by any of the sanity checks. This is the reason why we are confident about the correctness of our implementation.

1.4 Definitions and Notational Conventions

To describe our algorithms unambiguously we need some notation. We adopt notational conventions that are generally accepted by the integer programming and combinatorial optimisation community. People who are familiar with standard textbooks such as the ones by Nemhauser and Wolsey [88] and Grötschel, Lovász, and Schrijver [55] should recognise most of our notation.

Sets, Vectors. By $\mathbb{N}$ ($\mathbb{Z}, \mathbb{R}$) we denote the set of natural (integral, real) numbers. The natural numbers include 0. If S is a set, then the collection of all subsets of S is denoted by 2^S. If E and S are sets, where E is finite, then S^E is the set of vectors with $|E|$ components, where each component of a vector $\boldsymbol{x} \in S^E$ is indexed by an element of E, i.e., $\boldsymbol{x} = (x_e)_{e \in E}$. For $F \subseteq E$ the vector $\chi^F \in S^E$, defined by $\chi^F_e = 1$ if $e \in F$ and $\chi^F_e = 0$ if $e \in E \setminus F$, is called the *incidence vector* of F. For $F \subseteq E$ and $\boldsymbol{x} \in S^E$, the vector $\boldsymbol{x}_F \in S^F$ is the vector with $|F|$ components defined by $\boldsymbol{x}_F = (x_e)_{e \in F}$. We use $\boldsymbol{x}(F)$ to denote $\sum_{e \in F} x_e$. For $\boldsymbol{x} \in S^E$, the set $\operatorname{supp}(\boldsymbol{x}) = \{e \in E \mid x_e \neq 0\}$ is called the *support* of $\boldsymbol{x}$, and the set $\{e \in E \mid x_e \notin \mathbb{Z}\}$ is called the *fractional support*. The vectors $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$, defined by $x^+_e = x_e$ if $x_e \geq 0$, $x^+_e = 0$ if $x_e < 0$, $x^-_e = -x_e$ if $x_e \leq 0$, and $x^-_e = 0$ if $x_e > 0$, such that $\boldsymbol{x} = \boldsymbol{x}^+ - \boldsymbol{x}^-$, are called the positive part and the negative part of $\boldsymbol{x}$, respectively. All vectors are column vectors, unless stated otherwise.

Matrices. If I, J, and S are sets, where I and J are finite, then $S^{I \times J}$ is the set of $|I|$ by $|J|$ matrices, where each element of a matrix $A \in S^{I \times J}$ is indexed by an element from $I \times J$, i.e., $A = (a_{ij})_{i \in I, j \in J}$. Let $A \in S^{I \times J}$. For $I' \subseteq I, J' \subseteq J$, $A_{I'J'} \in S^{I' \times J'}$ denotes the $|I'| \times |J'|$ matrix defined by $(a_{ij})_{i \in I', j \in J'}$. For $j \in J$,

$A_j \in S^I$ denotes the j^{th} column of A, i.e., $A_j = A_{I\{j\}}$. For $i \in I$, $\boldsymbol{a}_i \in S^J$ denotes the i^{th} row of A written as a column vector, i.e., $\boldsymbol{a}_i = (A_{\{i\}J})^T$. For $J' \subseteq J$, $A_{J'} \in S^{I \times J'}$ denotes the $|I|$ by $|J'|$ matrix defined by $A_{J'} = A_{IJ'}$. For $I' \subseteq I$, $a_{I'} \in S^{I' \times J}$ denotes the $|I'|$ by $|J|$ matrix defined by $a_{I'} = A_{I'J}$. If A is invertible, the inverse of A is denoted by A^{-1}.

Undirected Graphs, Walks. An *(undirected) graph* $G = (V, E)$ consists of a finite nonempty set V of nodes and a finite set E of edges. Each edge $e \in E$ is an unordered pair $e = \{u, v\}$, u and v are called the *endpoints* of e. For each $S \subseteq V$, let $\delta(S) = \{\{u, v\} \mid u \in S, v \in V \setminus S\}$ be the set of edges that have exactly one endpoint in S. For $v \in V$, we write $\delta(v)$ instead of $\delta(\{v\})$. Given a subset $S \subseteq V$ of nodes, we use $E(S) = \{\{u, v\} \in E \mid u, v \in S\}$ to denote the set of edges with both endpoints in S. The graph with node set S and edge set $E(S)$ is called the *induced graph* of S and is denoted by $G[S] = (S, E(S))$.

A *walk* from v_0 to v_k in G is a finite sequence of nodes and edges $W = v_0, e_1, v_1, \ldots, e_k, v_k$ $(k \geq 0)$ such that for $i = 1, 2, \ldots, k$, $e_i = \{v_{i-1}, v_i\} \in E$. Node v_0 is the called the *start* of W and node v_k is called the *end* of W. The nodes on W are denoted by $V(W) = \{v_0, v_1, \ldots, v_k\}$, and the edges on W are denoted by $E(W) = \{e_1, e_2, \ldots, e_k\}$. The nodes $\{v_1, v_2, \ldots, v_{k-1}\}$ are called the *internal nodes* of W.

Directed Graphs, Walks. A directed graph $G = (V, A)$ consists of a finite nonempty set V of nodes and a finite set A of arcs. Each arc $a \in A$ is an ordered pair $a = (u, v)$, u and v are the *endpoints* of a, u is called the *tail* of a, and v the *head*. If $a = (u, v) \in A$ is an arc, we will denote the *reverse* arc of a by a^{-1}, i.e., $a^{-1} = (v, u)$. A directed graph is said to be *complete* if $A = \{(u, v) \mid u, v \in V, u \neq v\}$. For each $S \subseteq V$, denote by $\delta^+(S) = \{(u, v) \mid u \in S, v \in V \setminus S\}$ $(\delta^-(S) = \{(v, u) \mid u \in S, v \in V \setminus S\})$ the set of arcs leaving (entering, respectively) S. For $v \in V$, we write $\delta^+(v)$ $(\delta^-(v))$ instead of $\delta^+(\{v\})$ $(\delta^-(\{v\})$, respectively). Given a subset $S \subseteq V$ of nodes, we use $A(S) = \{(u, v) \in A \mid u, v \in S\}$ to denote the set of arcs with both endpoints in S. The directed graph with node set S and arc set $A(S)$ is again called the *induced graph* of S and is denoted by $G[S] = (S, A(S))$.

A *walk* from v_0 to v_k in G is a finite sequence of nodes and arcs $W = v_0, a_1, v_1, \ldots, a_k, v_k$ $(k \geq 0)$ such that for $i = 1, 2, \ldots, k$, $a_i = (v_{i-1}, v_i)$. Node v_0 is the called the *start* of W and node v_k is called the *end* of W. The nodes on W are denoted by $V(W) = \{v_0, v_1, \ldots, v_k\}$. The arcs on W are denoted by $A(W) = \{a_1, a_2, \ldots, a_k\}$. The nodes $\{v_1, v_2, \ldots, v_{k-1}\}$ are called the *internal nodes* of W.

Paths, Cycles, Holes. Suppose we are given an undirected (directed) graph $G = (V, E)$ $(G = (V, A))$. A *path* in G is a walk in G in which all nodes are distinct. We will denote a path from node u to node v by $u \rightsquigarrow v$. A *cycle* (*directed cycle*) in G is a walk in G with $v_0 = v_k$ in which all internal nodes are distinct and different from v_0. A *chord* in a cycle (directed cycle) C is an edge

$\{u, v\} \in E$ (arc $(u, v) \in A$) with $u, v \in V(C)$, but $\{u, v\} \notin E(C)$ ($(u, v) \notin E(C)$, respectively). A *hole* in G is a cycle in G without chords.

Part I

Foundations

Chapter 2

Branch-and-Bound and Linear Programming

2.1 Introduction

A central goal of this thesis is to study the behaviour of linear programming (LP) based branch-and-bound algorithms applied to a number of specific optimisation problems. Although the details of these problems will differ considerably from each other, they can all be formulated as integer linear programming problems (see also Section 1.1). Integer programming problems can be seen as instances of *optimisation problems*. We will formalise what we mean by optimisation problems in Section 2.2. When an optimisation problem satisfies some modest conditions then we can design a *branch-and-bound algorithm* for it, the subject of Section 2.3. A branch-and-bound algorithm can be seen as a divide-and-conquer algorithm which reduces the original optimisation problem we want to solve to a (possibly large) number of *relaxations* related to the original problem. In the case of linear integer programming problems these relaxations are linear programming problems. In Section 2.4 we review some selected topics from linear programming. This chapter gives the basis for Chapter 3, which combines branch-and-bound with linear programming to obtain an algorithm for integer linear programming problems.

2.2 Optimisation Problems

Although the problems that we will study in this thesis are very concrete, we will present the branch-and-bound core of our algorithms in terms of more abstract optimisation problems. Moreover, we will use the abstractions to relate our concrete problems to the theory of computational complexity. We start by defining optimisation problems in a way that suits our purposes.

Definition 2.1. An *optimisation problem* is defined by a collection of problem instances, and is either a *minimisation* problem or a *maximisation* problem. An *instance* of an optimisation problem is a pair (X, z), where X is the set of feasible solutions, and $z : X \to \mathbb{R}$ is the objective function.

Given an instance (X, z) of a maximisation (minimisation) problem, the problem is to find $x^* \in X$ such that $z(x^*) \geq z(x)$ (or $z(x^*) \leq z(x)$, respectively) for all $x \in X$.

Definition 2.2. An instance (X, z) of a maximisation (minimisation) problem is *feasible* if $X \neq \varnothing$, and *infeasible* otherwise. It is *bounded* if there exists an upper (lower) bound on the value of the objective function over the elements of X that is attained by a feasible solution, and *unbounded* otherwise.

To illustrate the intuition behind these definitions, consider the shortest path problem as an example. An instance of the shortest path problem is a pair $(\mathcal{P}, z)$ where $\mathcal{P}$ is given implicitly as the set of all paths in a graph $G = (V, E)$ that connect two nodes $s, t \in V$, and the objective function $z : \mathcal{P} \to \mathbb{R}$ is given by $z(P) = \boldsymbol{c}(E(P))$ for some cost vector $\boldsymbol{c} \in \mathbb{R}^E$. The shortest path problem is, given $(\mathcal{P}, z)$, to find a path $P \in \mathcal{P}$ that minimises $z(P)$. Since shortest paths are not necessarily unique, a given problem instance may have more than one optimal solution.

Definition 2.3. Let (X, z) be an instance of a maximisation (minimisation) problem P. A *relaxation* $\bar{P}$ of P is an associated maximisation (minimisation) problem, where each instance (X, z) of P is associated with an instance $(\bar{X}, \tilde{z})$ of $\bar{P}$ that satisfies

(i) $\bar{X} \supseteq X$, and

(ii) $\tilde{z}(x) \geq z(x)$ (or $\tilde{z}(x) \leq z(x)$, respectively) for all $x \in X$.

We assume that the reader has some intuitive feeling for what algorithms are. Informally, an algorithm for a problem is a well-defined computational procedure that takes a problem instance as input and produces a solution as output.

The notion of optimisation problem that is defined above suffices to introduce branch-and-bound algorithms. We will not be able to prove that the branch-and-bound algorithms that we propose are any good in terms of worst-case guarantees on the asymptotic behaviour of these algorithms. On the other hand they do solve the problems for which they are designed. In order to motivate our choice of algorithms, we use concepts from the theory of computational complexity. To link optimisation problems to the theory of computational complexity we need the following:

Definition 2.4. The *decision variant* of an optimisation problem is defined by a collection of problem instances. An *instance* to the decision variant of an optimisation problem P is a triple (X, z, ξ), where (X, z) is an instance of P, and $\xi \in \mathbb{R}$ a threshold value.

Given an instance (X, z, ξ) of the decision variant of a maximisation (minimisation) problem, the problem is to decide whether there exists a solution $x \in X$ such that $z(x) \geq \xi$ (or $z(x) \leq \xi$, respectively). In our shortest path example, the decision variant of the shortest path problem is, given an instance $(\mathcal{P}, z, \xi)$, to determine whether there exists a path $P \in \mathcal{P}$ such that $z(P) \leq \xi$.

Definitions 2.1–2.4 do not make any assumptions on the way a problem instance is specified. For optimisation problems of interest the set of feasible solutions is given implicitly, and instances are encoded using a *reasonable* encoding. The *size* of a problem instance is the length of its encoding. In our shortest path example a reasonable way to specify an instance is by giving a 4-tuple $(G, s, t, \boldsymbol{c})$, which has a size of $\Theta(|V| + |E|)$.

Define $\mathcal{P}$ to be the class of problems for which polynomial time algorithms do exist, and $\mathcal{NP}$ to be the class of problems for which solutions can be written down and verified in polynomial time of the size of the input. Cook's theorem (see, e.g. Papadimitriou [93]) proves the existence of so-called $\mathcal{NP}$*-complete* problems in $\mathcal{NP}$. These problems have the remarkable property that any polynomial time algorithm for it implies the existence of polynomial time algorithms for all problems in $\mathcal{NP}$, and thus that $\mathcal{P} = \mathcal{NP}$. So far no polynomial time algorithm has ever been developed for an $\mathcal{NP}$-complete problem. On the other hand, nobody has been able to show that $\mathcal{P} \neq \mathcal{NP}$. As a further introduction to computational complexity theory is beyond the scope of this thesis, we refer to Papadimitriou [93] for details. The relation between optimisation problems and their decision variants is treated in detail by Bovet and Crescenzi [21].

The decision variants of the optimisation problems we consider are all $\mathcal{NP}$-complete problems. Therefore we do not expect to be able to devise polynomial time algorithms for them. Instead we will study algorithms that either report feasible but not provably optimal solutions, or do not exhibit nice asymptotic behaviour. We even consider algorithms that neither report provably optimal solutions nor exhibit nice asymptotic behaviour. However, we will compare the quality of the solutions obtained with bounds on it, and we will establish at what size of problem instances this unfavourable asymptotic behaviour prevents us from finding solutions. This is done by means of computational experiments. We slightly abuse the theory of computational complexity by applying it to optimisation problems: we say that an optimisation problem is $\mathcal{NP}$*-hard* if its decision variant is $\mathcal{NP}$-complete.

2.3 Branch-and-Bound

Branch-and-bound algorithms started to appear in the literature around nineteen sixty. A survey of early branch-and-bound algorithms was given by Lawler and Wood [78]. Our description of the branch-and-bound algorithm is reminiscent of the one by Geoffrion [51], who uses Lagrangian relaxations.

Suppose we have an optimisation problem P and a relaxation $\bar{P}$ of P. Suppose further that for each instance (X, z) of P the associated instance $(\bar{X}, \bar{z})$ of $\bar{P}$ has the following properties:

```
branchAndBound(X, z)        // (X, z) is the instance of P we want to solve
{
    S := {X̄}; z* := −∞; i = 1;
    while S ≠ ∅ {
        extract X̄^i from S;
        solve (P̄^i): z̄^i = max{z̃(x) | x ∈ X̄^i};
        if (P̄^i) is feasible and z̄^i > z* {
            let x^i be the best available solution to (P̄^i);
            if x^i ∈ X and z(x^i) > z* {
                x* := x^i; z* := z(x^i);
                remove from S all X̄_1^j, X̄_2^j with z̄^j ≤ z*;
            }
            if z̄^i > z* {
                add X̄_1^i, X̄_2^i to S and store z̄^i together with X̄_1^i, X̄_2^i;
            }
        }
        i := i + 1;
    }
    if z* = −∞ { return infeasible; } else { return x*; }
}
```

Algorithm 2.1: The Branch-and-Bound Algorithm for Problem P

(i) Branching: we can partition each set $\bar{X}' \subseteq \bar{X}$ of feasible solutions to $\bar{P}$ into three disjoint sets $\bar{X}'_1, \bar{X}'_2$, and $\bar{X}' \setminus (\bar{X}'_1 \cup \bar{X}'_2)$ such that for all feasible solutions $x \in X \cap \bar{X}'$ to P we have either $x \in \bar{X}'_1$ or $x \in \bar{X}'_2$.

(ii) Bounding: for each set $\bar{X}' \subseteq \bar{X}$ of feasible solutions to $\bar{P}$ that can be obtained by branching we know how to solve the relaxation $(\bar{X}', \tilde{z})$ of the problem instance $(X \cap \bar{X}', z)$.

Assume that P is a maximisation problem and that P is either bounded or infeasible. Observe that

$$\max\{z(x) \mid x \in X \cap \bar{X}'\} \leq \max\{\tilde{z}(x) \mid x \in \bar{X}'\} \tag{2.1}$$

for all $\bar{X}' \subseteq \bar{X}$. Hence, property (ii) gives a way to bound the value of any feasible solution to P in $\bar{X}'$.

The *branch-and-bound* algorithm for P is defined as follows. The algorithm maintains a collection $\mathcal{S}$ of disjoint subsets from $\bar{X}$, the best known feasible solution $x^* \in X$, and its value $z^* = z(x^*)$, and works in iterations[1] starting

[1] A short summary of our notation: X denotes a set of feasible solutions to the problem of interest, $\bar{X}$ denotes a set of feasible solutions to its relaxation, X^i indicates a restricted set of feasible solutions to the problem of interest associated with iteration i, $\bar{X}^i$ denotes a set of feasible solutions to its relaxation, and the sets $\bar{X}^i_1$ and $\bar{X}^i_2$ are obtained from $\bar{X}^i$ by branching.

from iteration 1. Upon termination, if P is feasible then x^* will be an optimal solution to it. Initially, we take $\mathcal{S} = \{\bar{X}\}$, and set z^* to $-\infty$. In iteration i, we extract a set from $\mathcal{S}$, which we denote by $\bar{X}^i$, and solve the relaxation

$$\bar{z}^i = \max\{\tilde{z}(x) \mid x \in \bar{X}^i\}. \tag{2.2}$$

Note that we can do this because of property (ii), and that $\bar{z}^i$ is an upper bound on the value of any solution in $X \cap \bar{X}^i$ by (2.1). If problem (2.2) is infeasible, or if $\bar{z}^i \leq z^*$, we have a proof that there do not exist feasible solutions $x \in X \cap \bar{X}^i$ that have a larger objective function value than the best known solution and proceed with iteration $i+1$. Otherwise, let x^i be an optimal solution to problem (2.2). If x^i is a feasible solution to P and $z(x^i) > z^*$ we set $x^* = x^i$ and $z^* = z(x^i)$, and we remove from $\mathcal{S}$ all sets $\bar{X}^j_1$ and $\bar{X}^j_2$ with $\bar{z}^j \leq z^*$. If $\bar{z}^i > z^*$ we use property (i) to obtain disjoint sets $\bar{X}^i_1, \bar{X}^i_2$ that together contain all feasible solutions to P in $X \cap \bar{X}^i$ and add them to $\mathcal{S}$. Having done this we proceed with iteration $i + 1$. The algorithm terminates when $\mathcal{S}$ becomes empty. If upon termination $z^* = -\infty$ then the instance (X, z) is infeasible. Otherwise the algorithm returns x^*. Pseudo-code is given in Algorithm 2.1.

The correctness of the branch-and-bound algorithm is seen as follows. Denote the value of z^* at the end of iteration i by z^*_i, and the set $\mathcal{S}$ at the end of iteration i by $\mathcal{S}^i$. At the end of each iteration of the algorithm, each solution $x \in X$ that has a larger objective function value than the value z^*_i of the best known solution in iteration i is contained in some set $\bar{X}' \in \mathcal{S}^i$, i.e.,

$$\text{for all } x \in X \text{ with } z(x) > z^*_i \text{ there exists } \bar{X}' \in \mathcal{S}^i \text{ such that } x \in \bar{X}'. \tag{2.3}$$

Since upon termination $\mathcal{S}$ is empty and $z^* = z(x^*)$ if $z^* > -\infty$, either there does not exist a feasible solution or the reported solution x^* is optimal. The branch-and-bound algorithm terminates if for any sequence $\bar{X}^{i_1} \supseteq \bar{X}^{i_2} \supseteq \cdots$ we have that $\bar{X}^{i_k} \subseteq X$ for some finite index k. If n is an upper bound on any such index k, then the maximum number of relaxations solved is bounded by 2^{n-1}.

From (2.3) it follows that if $\mathcal{S}^i \neq \varnothing$, then the value

$$\max\{\bar{z}^i \mid \bar{X}^i_k \in \mathcal{S}^i\} \tag{2.4}$$

is an upper bound on the value of the optimal solution to P. This can be used to obtain a good upper bound if one terminates the algorithm before one completed the enumeration.

With every execution of a branch-and-bound algorithm we can associate a rooted tree T that we call the *branch-and-bound tree*. Consider any finite execution of the branch-and-bound algorithm and let the last iteration in this execution be iteration N. With any iteration i, we associate a node v_i in T ($i \in \{1, \ldots, N\}$); node v_1 is the root of the tree. For each iteration $j > 1$, we add the edge $\{v_i, v_j\}$, where v_i is uniquely determined by $\bar{X}^j = \bar{X}^i_k$ for some $k \in \{1, 2\}$. The branch-and-bound tree $T = (V, E)$, rooted at v_1, is defined by

$$V = \{v_i \mid i \in \{1, \ldots, N\}\}, \text{ and}$$
$$E = \{\{v_i, v_j\} \mid \bar{X}^j = \bar{X}^i_k, k \in \{1, 2\}, j \in \{2, \ldots, N\}\}.$$

There are two ingredients of the branch-and-bound algorithm that we left unspecified, namely, how to choose the relaxation $\bar{X}^i \in \mathcal{S}$ in iteration $i > 1$, which is called the *enumeration scheme*, and how to choose the relaxations $\bar{X}_1^i, \bar{X}_2^i$ after solving $\bar{X}^i$, which is called the *branching scheme*. Both schemes can influence the size of the branch-and-bound tree, and hence the running time of the branch-and-bound algorithm. Note that if we can guarantee for a certain problem P that the branch-and-bound tree has a polynomial size and that each iteration terminates in polynomial time, then the branch-and-bound algorithm is a polynomial time algorithm for P. Hence, a branch-and-bound algorithm for an $\mathcal{NP}$-hard problem in which the relaxations solved by the algorithm are in $\mathcal{P}$ is unlikely to have an enumeration scheme or a branching scheme that yields a polynomial size branch-and-bound tree. This explains why branching rules and enumeration schemes that are encountered in the literature all have a heuristic nature. Branching schemes are problem specific, so we postpone the discussion of them until we apply branch-and-bound to concrete problems. Although enumeration schemes can be problem specific as well, there are a number of enumeration schemes that can be mentioned within the abstract branch-and-bound framework, namely, the *depth-first*, *breadth-first*, and *best-first* enumeration schemes.

A depth-first enumeration scheme chooses the relaxations in such a way that the sequence of nodes $(v_i)_{i=1}^N$ is a depth-first traversal of T, where T is the associated branch-and-bound tree. The advantage of the depth-first enumeration scheme is that it steers towards the leaves of T, thus creating an opportunity to improve z^* (or, if we are unlucky, find many infeasible relaxations). During a depth-first traversal $|\mathcal{S}|$ is bounded by the maximum depth attained by any node in T, which make depth-first traversals efficient in terms of memory requirement. Another advantage of the depth-first enumeration scheme is that it can be implemented easily using a recursive algorithm. Such a recursive algorithm can take maximal advantage of similarities between the relaxations of adjacent nodes in the tree in order to decrease the computation time needed for each node. A disadvantage of the depth-first enumeration scheme is that the depth-first traversal can enter "the wrong part of the tree", that is, a subtree of T in which the relaxations associated with the nodes of the subtree do not contain the optimal solution, or even worse, do not contain feasible solutions $x \in X$. The branch-and-bound algorithm will not find this out until it has completed the computation in the subtree.

The breadth-first enumeration scheme chooses the relaxations in such a way that the sequence of nodes $(v_i)_{i=1}^N$ is a breadth-first traversal of T. It does not have the disadvantage of the depth-first enumeration scheme that it can digress into the wrong part of the branch-and-bound tree. It does have the disadvantage that it can take quite a long time before it encounters feasible solutions $x \in X$.

The best-first enumeration scheme chooses the relaxation with the best upper bound, i.e., in iteration $i > 1$ it chooses $\bar{X}^i = \arg\max_{\bar{X}_k^j \in \mathcal{S}^i} \bar{z}^j$. Observe that at the end of iteration i, the value of $\max_{\bar{X}_k^j \in \mathcal{S}^i} \bar{z}^j$ is an upper bound on the value of any solution $x \in X$ that is contained in some relaxation $\bar{X} \in \mathcal{S}^i$,

including all optimal solutions to the problem at hand. The advantage of the best-first enumeration scheme is that it never chooses to solve a relaxation that does not contribute to proving the optimality of these optimal solutions. On the other hand, the best-first enumeration scheme has the same disadvantage as the breadth-first enumeration scheme, in that it can take quite long before it encounters feasible solutions $x \in X$. In our computational experiments we will use the best-first enumeration scheme.

We end our discussion of branch-and-bound with two observations. The first observation is that it does not matter for the correctness of the algorithm in what way we obtain our best feasible solution x^*. Therefore, if we have some heuristic for P that gives us a feasible $x \in X$, we can use it to initialise x^* and z^*. Moreover, if we have a heuristic for P that maps solutions $x \in \bar{X}$ to $x' \in X$, and this heuristic is reasonably fast, then we can apply it to the solution x^i in each iteration i for which $x^i \notin X$, and possibly improve upon the value of z^*. The second observation is that we can turn the branch-and-bound algorithm into a heuristic by just stopping it after a fixed time or iteration limit. If, at the time we stop the algorithm, we have some feasible solution x^*, we take this as the result. Otherwise, our heuristic has failed to find a solution.

2.4 Linear Programming

Linear programming was developed by Dantzig [33] in 1947. Textbooks on linear programming have been written, among others, by Dantzig [34], Chvátal [28], Bazaraa, Jarvis, and Sherali [16], Vanderbei [116], and Dantzig and Thapa [35]. In this section we follow the exposition of Dantzig and Thapa ([35, Section 3.4]). Recall that the rank of a matrix A is the number of linearly independent rows (or columns) of A. An $m \times n$ matrix A (with $m \leq n$) is of full rank if the rank of A is m.

Definition 2.5. Given natural numbers m, n, vectors $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$, and a matrix $A \in \mathbb{R}^{m \times n}$, the *linear programming problem* is to determine

$$\begin{aligned} \max \quad & \boldsymbol{c}^T \boldsymbol{x} && \text{(2.5a)} \\ \text{subject to} \quad & A\boldsymbol{x} = \boldsymbol{b} && \text{(2.5b)} \\ & \boldsymbol{x} \geq \boldsymbol{0}. && \text{(2.5c)} \end{aligned}$$

Our interest will be mainly in the following variant of the linear programming problem.

Definition 2.6. Given natural numbers m, n, vectors $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{l}, \boldsymbol{u} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$, and a matrix $A \in \mathbb{R}^{m \times n}$, the *linear programming problem with bounded variables* is to determine

$$\begin{aligned} \max \quad & \boldsymbol{c}^T \boldsymbol{x} && \text{(2.6a)} \\ \text{subject to} \quad & A\boldsymbol{x} = \boldsymbol{b} && \text{(2.6b)} \\ & \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}. && \text{(2.6c)} \end{aligned}$$

The function $z(\boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x}$ is called the *objective function*. In the remainder of this section we restrict our attention to linear programming problems with bounded variables.

Definition 2.7. A vector $\boldsymbol{x} \in \mathbb{R}^n$ is called a *feasible solution* to (2.6) if it satisfies (2.6b)–(2.6c).

In the following, to simplify the discussion we assume that A is of full rank.

Definition 2.8. A *basis* of A is a set $\{A_{j_1}, A_{j_2}, \ldots, A_{j_m}\}$ of linearly independent vectors.

We will slightly overload terminology by calling the set of column indices $\{j_1, j_2, \ldots, j_m\}$ a basis if $\{A_{j_1}, A_{j_2}, \ldots, A_{j_m}\}$ is a basis. Denote the set of column indices by $J = \{1, \ldots, n\}$. The *basic variables* corresponding to a basis $B \subseteq J$ are the variables $\{x_j \mid j \in B\}$. If $B \subseteq J$ is a basis of A, then A_B is an invertible matrix. Suppose we have a basis $B \subseteq J$.

Definition 2.9. The vector $\boldsymbol{x} \in \mathbb{R}^n$ is a *basic solution* corresponding to the basis $B \subseteq J$ if we can partition $J \setminus B$ into two disjoint sets $J_1, J_2 \subseteq J \setminus B$ (with $J_1 \cup J_2 = J \setminus B$) such that

$$\boldsymbol{x}_{J_1} = \boldsymbol{l}_{J_1}, \tag{2.7a}$$

$$\boldsymbol{x}_{J_2} = \boldsymbol{u}_{J_2}, \text{ and} \tag{2.7b}$$

$$\boldsymbol{x}_B = A_B^{-1}(\boldsymbol{b} - A_{J\setminus B}\boldsymbol{x}_{J\setminus B}). \tag{2.7c}$$

So, in a basic solution the variables $\boldsymbol{x}_{J_1}$ are at their lower bound, the variables $\boldsymbol{x}_{J_2}$ are at their upper bound, and the values of the basic variables are solved for.

Definition 2.10. Given a vector $\boldsymbol{\pi} \in \mathbb{R}^m$, the *reduced cost* with respect to $\boldsymbol{\pi}$ is defined as the vector $\boldsymbol{c}^{\boldsymbol{\pi}} \in \mathbb{R}^n$ that is given by $\boldsymbol{c}^{\boldsymbol{\pi}} = \boldsymbol{c} - (\boldsymbol{\pi}^T A)^T$.

A fundamental result from the theory of linear programming is that, given a basis B, we can rewrite the objective function in terms of non-basic variables:

Proposition 2.1. *Let $A \in \mathbb{R}^{m\times n}, \boldsymbol{b} \in \mathbb{R}^m, \boldsymbol{c} \in \mathbb{R}^n$, and let $B \subseteq J$ be a basis of A. If $\boldsymbol{x}$ satisfies (2.6b), then*

$$\boldsymbol{c}^T\boldsymbol{x} = \boldsymbol{\pi}^T\boldsymbol{b} + (\boldsymbol{c}^{\boldsymbol{\pi}}_{J\setminus B})^T\boldsymbol{x}_{J\setminus B},$$

where $\boldsymbol{\pi}^T = \boldsymbol{c}_B^T A_B^{-1}$.

Proof. Assume that $\boldsymbol{x}$ satisfies (2.6b). It follows that $\boldsymbol{x}$ also satisfies (2.7c). Hence,

$$\begin{aligned}
\boldsymbol{c}^T\boldsymbol{x} &= \boldsymbol{c}_B^T\boldsymbol{x}_B + \boldsymbol{c}_{J\setminus B}^T\boldsymbol{x}_{J\setminus B} \\
&= \boldsymbol{c}_B^T(A_B^{-1}(\boldsymbol{b} - A_{J\setminus B}\boldsymbol{x}_{J\setminus B})) + \boldsymbol{c}_{J\setminus B}^T\boldsymbol{x}_{J\setminus B} \\
&= \boldsymbol{c}_B^T A_B^{-1}\boldsymbol{b} - \boldsymbol{c}_B^T A_B^{-1} A_{J\setminus B}\boldsymbol{x}_{J\setminus B} + \boldsymbol{c}_{J\setminus B}^T\boldsymbol{x}_{J\setminus B} \\
&= \boldsymbol{\pi}^T\boldsymbol{b} + (\boldsymbol{c}_{J\setminus B}^T - \boldsymbol{c}_B^T A_B^{-1} A_{J\setminus B})\boldsymbol{x}_{J\setminus B} \\
&= \boldsymbol{\pi}^T\boldsymbol{b} + (\boldsymbol{c}^{\boldsymbol{\pi}}_{J\setminus B})^T\boldsymbol{x}_{J\setminus B}.
\end{aligned}$$

□

Theorem 2.2. (Reduced Cost Optimality Conditions) *Let $\boldsymbol{x}^* \in \mathbb{R}^n$ be a feasible solution to (2.6), $B \subseteq J$ a basis of A, and let $\boldsymbol{\pi}^T = \boldsymbol{c}_B^T A_B^{-1}$. If $(\boldsymbol{x}^*, \boldsymbol{\pi})$ satisfies*

$$\begin{aligned} &c_j^{\boldsymbol{\pi}} < 0 \text{ implies } x_j^* = l_j, \text{ and} \\ &c_j^{\boldsymbol{\pi}} > 0 \text{ implies } x_j^* = u_j, \end{aligned} \qquad \text{for all } j \in J, \tag{2.8}$$

then $\boldsymbol{x}^*$ is an optimal solution to (2.6).

Proof. Assume that $(\boldsymbol{x}^*, \boldsymbol{\pi})$ satisfies (2.8), and let $\boldsymbol{x}$ be an arbitrarily chosen feasible solution to (2.6). Let $J_1 = \{j \in J \mid c_j^{\boldsymbol{\pi}} < 0\}$ and $J_2 = \{j \in J \mid c_j^{\boldsymbol{\pi}} > 0\}$. Note that $\boldsymbol{c}_B^{\boldsymbol{\pi}} = \boldsymbol{0}$, so $J_1, J_2 \subseteq J \setminus B$. By (2.8) and feasibility of $\boldsymbol{x}$ we have that $\boldsymbol{x}^*_{J_1} \leq \boldsymbol{x}_{J_1}$ and $\boldsymbol{x}^*_{J_2} \geq \boldsymbol{x}_{J_2}$. By applying Proposition 2.1, we find that

$$\begin{aligned} \boldsymbol{c}^T\boldsymbol{x}^* - \boldsymbol{c}^T\boldsymbol{x} &= (\boldsymbol{c}^{\boldsymbol{\pi}}_{J\setminus B})^T\boldsymbol{x}^*_{J\setminus B} - (\boldsymbol{c}^{\boldsymbol{\pi}}_{J\setminus B})^T\boldsymbol{x}_{J\setminus B} \\ &= (\boldsymbol{c}^{\boldsymbol{\pi}}_{J\setminus B})^T(\boldsymbol{x}^*_{J\setminus B} - \boldsymbol{x}_{J\setminus B}) \\ &= (\boldsymbol{c}^{\boldsymbol{\pi}}_{J_1})^T(\boldsymbol{x}^*_{J_1} - \boldsymbol{x}_{J_1}) + (\boldsymbol{c}^{\boldsymbol{\pi}}_{J_2})^T(\boldsymbol{x}^*_{J_2} - \boldsymbol{x}_{J_2}) \\ &\geq 0, \end{aligned}$$

where the inequality is due to condition (2.8). Hence, $\boldsymbol{c}^T\boldsymbol{x}^* \geq \boldsymbol{c}^T\boldsymbol{x}$. Because $\boldsymbol{x}$ was chosen arbitrarily, $\boldsymbol{x}^*$ is optimal. □

We refer to the process of determining whether (2.6) is infeasible, unbounded, or has a solution $(\boldsymbol{x}^*, \boldsymbol{\pi})$ satisfying the reduced-cost optimality conditions as *solving* (2.6).

Consider the following linear programming problem related to (2.5):

$$\begin{aligned} \min \quad & \boldsymbol{b}^T\boldsymbol{\pi} && \text{(2.9a)} \\ \text{subject to} \quad & A^T\boldsymbol{\pi} \geq \boldsymbol{c} && \text{(2.9b)} \\ & \boldsymbol{\pi} \lesseqgtr \boldsymbol{0}. && \text{(2.9c)} \end{aligned}$$

A fundamental theorem from the theory of linear programming is the following:

Theorem 2.3. (Strong Duality Theorem) *If (2.5) is unbounded then (2.9) is infeasible. If (2.9) is unbounded then (2.5) is infeasible. If (2.5) and (2.9) are both feasible then $\boldsymbol{c}^T\boldsymbol{x}^* = \boldsymbol{b}^T\boldsymbol{\pi}^*$, where $\boldsymbol{x}^*$ and $\boldsymbol{\pi}^*$ are optimal solutions to (2.5) and (2.9), respectively.*

The problem (2.9) is called the *dual problem* of (2.5). For further details of the theory of linear programming (including LP duality) we refer to the books mentioned at the beginning of this chapter.

The original method proposed by Dantzig for solving linear programming problems is called *the simplex method*. The simplex method assumes that it has as input a basis $B \subseteq J$ with a corresponding basic feasible solution, that it uses as a starting point. A variant of the simplex method, called the *dual simplex method*, assumes that it has as input a basis $B \subseteq J$ for which the

vector $\boldsymbol{\pi} = (\boldsymbol{c}_B^T A_B^{-1})^T$ is part of a feasible solution to a linear program that is dual to (2.6) (see Vanderbei [116, Chapter 9]). Although no polynomial time bound has been shown to hold for any version of the simplex method, the algorithm performs very well in practice. A basic feasible solution to any linear programming problem can be found, if it exists, by solving a derived linear programming problem with an obvious starting basis, at the cost of some extra computation time. There are various codes available that implement the simplex method. In our implementations we make use of the CPLEX 6.5 linear programming solver [62].

It was shown by Khachiyan [69] that linear programming problems can be solved in polynomial time using an algorithm called *the ellipsoid method* (see e.g. Grötschel, Lovász, and Schrijver [55]). The importance of this algorithm is that it proves that linear programming is in $\mathcal{P}$.

Chapter 3

LP-Based Branch-and-Bound

3.1 Introduction

We have seen in Section 2.3 how to design branch-and-bound algorithms for optimisation problems. In this thesis our focus is not on optimisation problems in general, but on specific problems that are the subject of Part II. Each problem discussed in Part II of this thesis can be formulated as a *mixed integer linear programming problem*. A mixed integer linear programming problem can be defined as follows. Given a matrix $A \in \mathbb{Z}^{m \times n}$, vectors $\boldsymbol{c}, \boldsymbol{l}, \boldsymbol{u} \in \mathbb{Z}^n$ and $\boldsymbol{b} \in \mathbb{Z}^m$, and a subset of the column indices $J \subseteq \{1, \ldots, n\}$, find

$$\begin{aligned} \max \quad & z(\boldsymbol{x}) = \boldsymbol{c}^T \boldsymbol{x} && \text{(3.1a)} \\ \text{subject to} \quad & A\boldsymbol{x} = \boldsymbol{b}, && \text{(3.1b)} \\ & \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}, && \text{(3.1c)} \\ & \boldsymbol{x}_J \text{ integer}. && \text{(3.1d)} \end{aligned}$$

The special case with $J = \{1, \ldots, n\}$ is called a *(pure) integer linear programming problem*. If in an integer linear program all variables are allowed to take values zero or one only, then it is called a *zero-one* integer linear programming problem. Let

$$P = \{\boldsymbol{x} \in \mathbb{R}^n \mid A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\},$$

and

$$X = P \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{x}_J \text{ integer}\}.$$

Problem (3.1) is equivalent to

$$\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in X\},$$

which is an instance of an optimisation problem in the sense of Definition 2.1. The *LP relaxation* of problem 3.1 is obtained by removing the constraints (3.1d), which yields the problem

$$\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in P\}. \tag{3.2}$$

In this chapter we will study branch-and-bound algorithms that take advantage of the fact that LP relaxations can be solved efficiently. In case the entire formulation can be stored in the main memory of the computer, one can apply the basic LP-based branch-and-bound algorithm. However, it may occur that A is given implicitly, and in this case m and n can even be exponential in the size of a reasonable encoding of the problem data. Sometimes we are still able to handle such problems using some modification of the basic LP-based branch-and-bound algorithm. These modifications lead to so-called branch-and-cut, branch-and-price, and branch-price-and-cut algorithms, corresponding to the cases in which only a subset of the constraints, a subset of the variables, and both a subset of the constraints and of the variables are kept in main memory, respectively.

The area of integer linear programming was pioneered by Gomory [53] who developed his famous cutting plane algorithm for integer linear programming problems in the nineteen fifties. Two of the first branch-and-bound algorithms for integer linear programming were developed by Land and Doig [74] and Dakin [32] in the early nineteen sixties. Since that time, many articles, books and conferences have been devoted to the subject. Among the books on integer programming we mention Papadimitriou and Steiglitz [94], Schrijver [104], Nemhauser and Wolsey [88], and Wolsey [125]. Recent surveys of branch-and-bound algorithms for integer programming are by Jünger, Reinelt and Thienel [67] and Johnson, Nemhauser, and Savelsbergh [66]. Together with the papers by Padberg and Rinaldi [92] and Linderoth and Savelsbergh [79], these references are the main source of the ideas that led to the algorithms in this chapter. Recently, approaches to integer linear programming that are different from LP-based branch-and-bound have been reported on by Aardal, Weismantel and Wolsey [3].

The advances of the theory and the developments in computer hardware and software during the past four decades have resulted in algorithms that are able to solve relevant integer programming problems in practice to optimality. This makes linear integer programming an important subject to study.

Several software packages that allow for the implementation of customised branch-and-cut, branch-and-price, and branch-price-and-cut algorithms exist. Here we mention MINTO [84] and ABACUS [110]. In order to have full freedom of algorithmic design we have implemented our own framework for branch-and-cut, branch-and-price, and branch-price-and-cut, which we use in all computational experiments that are reported on in this thesis. The goal of this chapter is to give the reader the opportunity to find out what we actually implemented, without having to make a guess based upon the literature we refer to.

The remainder of this chapter is organised as follows. We describe a basic LP-based branch-and-bound algorithm in Section 3.2. We describe our version

of the branch-and-cut, branch-and-price, and branch-price-and-cut algorithms in Sections 3.3, 3.4, and 3.5, respectively.

3.2 An LP-Based Branch-and-Bound Algorithm

Here we refine the branch-and-bound algorithm from Section 2.3 for linear integer programming. The relaxations solved in a node of the branch-and-bound tree are given in Section 3.2.1. Sometimes it is possible to improve the linear formulation of the problem in a part of the branch-and-bound tree by tightening the bounds on the variables. This is discussed in Sections 3.2.2 and 3.2.3. We proceed by describing branching schemes that can be employed in Sections 3.2.4 and 3.2.5.

3.2.1 LP Relaxations

Consider iteration i of the branch-and-bound algorithm. The LP relaxation we solve in iteration i of the branch-and-bound algorithm is uniquely determined by its lower and upper bounds on the variables, which we will denote by $\boldsymbol{l}^i$ and $\boldsymbol{u}^i$, respectively. Let

$$P^i = \{\boldsymbol{x} \in \mathbb{R}^n \mid A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{l}^i \leq \boldsymbol{x} \leq \boldsymbol{u}^i\},$$

and

$$X^i = \{\boldsymbol{x} \in P^i \mid \boldsymbol{x}_J \text{ integer}\}.$$

The LP relaxation that we solve in iteration i, denoted by LP^i, is given by

$$\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in P^i\}, \tag{3.3}$$

which is a linear program with bounded variables as discussed in Section 2.4. In the root node v_1 we take $\boldsymbol{l}^1 = \boldsymbol{l}$ and $\boldsymbol{u}^1 = \boldsymbol{u}$ to obtain the LP relaxation (3.2) of the original problem (3.1).

Note that the matrix A is a constant. In an implementation of the LP-based branch-and-bound algorithm this can be exploited by maintaining only one LP formulation of the problem. When formulating LP^i in iteration i of the branch-and-bound algorithm, we do this by imposing the bounds $\boldsymbol{l}^i, \boldsymbol{u}^i$ in this formulation.

Next, we keep track of the basis associated with the optimal solution to the LP relaxation that we solve in each node of the branch-and-bound tree. How we do this is explained in more detail in Section 3.2.2. Recall the construction of the branch-and-bound tree from Section 2.3. Consider iteration $i > 1$ of the branch-and-bound algorithm. The optimal basis $B \subseteq \{1, \ldots, n\}$ associated with the parent of node v_i in the branch-and-bound tree defines a dual solution $\boldsymbol{\pi} = (\boldsymbol{c}_B^T A_B^{-1})^T$. Furthermore LP^i is derived from the LP relaxation associated with the parent of node v_i in the branch-and-bound tree by modifying only a small number of variable bounds. Therefore $\boldsymbol{\pi}$ is dual feasible to LP^i, and we can expect $\boldsymbol{\pi}$ to be close to optimal. This is exploited by solving the LP^i starting from $\boldsymbol{\pi}$ using the dual simplex algorithm.

3.2.2 Tightening Variable Bounds and Setting Variables

Suppose we have at our disposal a vector $\boldsymbol{x}^* \in X$ with $z(\boldsymbol{x}^*) = z^*$. Consider iteration i of the branch-and-bound algorithm in which LP^i was feasible, and suppose we have solved it to optimality. Recall that our branch-and-bound algorithm is correct as long as we do not discard any solution that is better than our current best solution from the remaining search-space (or, more precisely, if we maintain condition (2.3) as invariant). We can exploit the information obtained from the optimal solution of LP^i to tighten the bounds $\boldsymbol{l}^i$ and $\boldsymbol{u}^i$. The improved bounds are based on the value z^* and the reduced cost of non-basic variables in an optimal LP solution.

Let $(\boldsymbol{x}^{\mathrm{LP}}, \boldsymbol{\pi})$ be an optimal primal-dual pair to LP^i, where $\boldsymbol{\pi} = (\boldsymbol{c}_B^T A_B^{-1})^T$ for some basis $B \subseteq \{1, \ldots, n\}$. Further, let $z^{\mathrm{LP}} = z(\boldsymbol{x}^{LP})$, and let $L, U \subseteq \{1, \ldots, n\}$ be the sets of variable indices with $\boldsymbol{c}_L^{\boldsymbol{\pi}} < \boldsymbol{0}$ and $\boldsymbol{c}_U^{\boldsymbol{\pi}} > \boldsymbol{0}$. The reduced cost $c_j^{\boldsymbol{\pi}}$ can be interpreted as the change of the objective function per unit change of variable x_j. From the reduced cost optimality conditions (see Theorem 2.2) it follows that $x_j = u_j$ if $c_j^{\boldsymbol{\pi}} > 0$ and $x_j = l_j$ if $c_j^{\boldsymbol{\pi}} < 0$. Using these observations and the difference in objective function between the optimal LP solution and $\boldsymbol{x}^*$ we can compute a new lower bound for x_j if $c_j^{\boldsymbol{\pi}} > 0$, and a new upper bound for x_j if $c_j^{\boldsymbol{\pi}} < 0$. These improved bounds are given by $\tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i \in \mathbb{Q}^n$, where

$$\tilde{l}_j^i = \begin{cases} \max(l_j^i, u_j^i + \lceil (z^* - z^{\mathrm{LP}})/c_j^{\boldsymbol{\pi}} \rceil), & \text{if } j \in U \cap J, \\ \max(l_j^i, u_j^i + (z^* - z^{\mathrm{LP}})/c_j^{\boldsymbol{\pi}}), & \text{if } j \in U \setminus J, \\ l_j^i & \text{otherwise,} \end{cases}$$

and

$$\tilde{u}_j^i = \begin{cases} \min(u_j^i, l_j^i + \lfloor (z^* - z^{\mathrm{LP}})/c_j^{\boldsymbol{\pi}} \rfloor), & \text{if } j \in L \cap J, \\ \min(i_j^i, l_j^i + (z^* - z^{\mathrm{LP}})/c_j^{\boldsymbol{\pi}}), & \text{if } j \in L \setminus J, \\ u_j^i, & \text{otherwise.} \end{cases}$$

The following proposition proves the correctness of the improved bounds.

Proposition 3.1. *All $\boldsymbol{x}^{IP} \in X^i$ with $z(\boldsymbol{x}^{IP}) \geq z^*$ satisfy $\tilde{\boldsymbol{l}}^i \leq \boldsymbol{x}^{IP} \leq \tilde{\boldsymbol{u}}^i$.*

Proof. Let $\boldsymbol{x}^{\mathrm{LP}}, \boldsymbol{\pi}, z^{\mathrm{LP}}, L, U$ be as in the construction of $z^*, i, \tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i$. Assume that there exists a vector $\boldsymbol{x}^{\mathrm{IP}} \in X^i$ with $z(\boldsymbol{x}^{\mathrm{IP}}) \geq z^*$. Since $\boldsymbol{x}^{\mathrm{IP}} \in X^i \subseteq P^i$ we have $A\boldsymbol{x}^{\mathrm{IP}} = \boldsymbol{b}$, so by Proposition 2.1 we can write

$$z(\boldsymbol{x}^{\mathrm{IP}}) = \boldsymbol{\pi}^T \boldsymbol{b} + (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T \boldsymbol{x}_L^{\mathrm{IP}} + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T \boldsymbol{x}_U^{\mathrm{IP}},$$

and since $\boldsymbol{x}^{\mathrm{LP}} \in P^i$ we can write

$$\begin{aligned} z(\boldsymbol{x}^{\mathrm{LP}}) &= \boldsymbol{\pi}^T \boldsymbol{b} + (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T \boldsymbol{x}_L^{\mathrm{LP}} + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T \boldsymbol{x}_U^{\mathrm{LP}} \\ &= \boldsymbol{\pi}^T \boldsymbol{b} + (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T \boldsymbol{l}_L^i + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T \boldsymbol{u}_U^i. \end{aligned}$$

Observe that $\boldsymbol{x}^{\mathrm{IP}} \in P_i$ implies $\boldsymbol{x}^{\mathrm{IP}}_L \geq \boldsymbol{l}^i_L$ and $\boldsymbol{x}^{\mathrm{IP}}_U \leq \boldsymbol{u}^i_U$, so $\boldsymbol{x}^{\mathrm{IP}}_L - \boldsymbol{l}^i_L \geq \mathbf{0}$ and $\boldsymbol{x}^{\mathrm{IP}}_U - \boldsymbol{u}^i_U \leq \mathbf{0}$. Now, choose $j \in U$ arbitrarily. Note that $\boldsymbol{x}^{\mathrm{IP}} \in X^i \subseteq P^i$ directly gives $x^{\mathrm{IP}}_j \geq l^i_j$. Moreover,

$$\begin{aligned} z^* - z^{\mathrm{LP}} &\leq z(\boldsymbol{x}^{\mathrm{IP}}) - z(\boldsymbol{x}^{\mathrm{LP}}) \\ &= (\boldsymbol{c}^{\boldsymbol{\pi}}_L)^T(\boldsymbol{x}^{\mathrm{IP}}_L - \boldsymbol{l}^i_L) + (\boldsymbol{c}^{\boldsymbol{\pi}}_U)^T(\boldsymbol{x}^{\mathrm{IP}}_U - \boldsymbol{u}^i_U) \\ &\leq c^{\boldsymbol{\pi}}_j(x^{\mathrm{IP}}_j - u^i_j). \end{aligned}$$

Hence,

$$x^{\mathrm{IP}}_j \geq u^i_j + (z^* - z^{\mathrm{LP}})/c^{\boldsymbol{\pi}}_j.$$

If $j \in U \setminus J$ this proves that $x^{\mathrm{IP}}_j \geq \tilde{l}^i_j$. Otherwise, $j \in U \cap J$, and $x^{\mathrm{IP}}_j \geq \tilde{l}^i_j$ by integrality of x^{IP}_j. Because j was chosen arbitrarily we have $\boldsymbol{x}^{\mathrm{IP}} \geq \tilde{\boldsymbol{l}}^i$. The proof that $\boldsymbol{x}^{\mathrm{IP}} \leq \tilde{\boldsymbol{u}}^i$ is derived similarly starting from an arbitrarily chosen index $j \in L$. □

Denote the sub-tree of the branch-and-bound tree that is rooted at node v_i by T_{v_i}. We can tighten the bounds on the variables after solving the LP in iteration i by replacing the bounds $\boldsymbol{l}^i, \boldsymbol{u}^i$ by $\tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i$. By Proposition 3.1 we do not discard any solution satisfying the integrality conditions that is better than the current best solution $\boldsymbol{x}^*$ in doing so, which means that we maintain condition (2.3) as invariant. The improved bounds are used in all iterations of the branch-and-bound algorithm that are associated with a node in the branch-and-bound tree in the sub-tree rooted at node v_i, the node in the branch-and-bound tree associated with iteration i. In our implementation of LP-based branch-and-bound we do not tighten the bounds on continuous variables.

When a variable index $j \in \{1, \ldots, n\}$ satisfies $l^i_j = u^i_j$ we say that x_j is *set* to $l^i_j = u^i_j$ in iteration i (node v_i). When a variable is set in the root node of the branch-and-bound tree, it is called *fixed*. If $l^i_j < u^i_j$, we say that that x_j is *free* in iteration i (node v_i). Variable setting based on reduced cost belongs to the folklore and is used by many authors to improve the formulation of zero-one integer programming problems (for example by Crowder, Johnson, and Padberg [31] and Padberg and Rinaldi [92]). The version in Proposition 3.1 is similar to the one mentioned by Wolsey [125, Exercise 7.8.7].

Note that the new bounds are a function of $\boldsymbol{\pi}, z^{\mathrm{LP}}$, and z^*. As a consequence, each time that we find a new primal solution in the branch-and-bound algorithm we can re-compute the bounds. Suppose we find an improved primal solution in iteration k. An original feature of our implementation of the branch-and-bound algorithm is that we re-compute the bounds in all nodes v_i of the branch-and-bound tree with $i \in \{1, \ldots, k\}$ that are on a path from the root node to a node $v_{k'}$ with $k' > k$. In order to be able to do this, we store a tree T' that mirrors the branch-and-bound tree. Each node w_i in T' corresponds to some iteration i of the branch-and-bound algorithm, and with w_i we store its parent $p(w_i)$ in T', and the values of $\boldsymbol{\pi}$, z^{LP}, and z^* for which we last computed the bounds

in w_i, and the bounds that we can actually improve in node w_i. The values of $\boldsymbol{\pi}$ are stored implicitly by storing only the differences of the optimal LP basis between node w_i and node $p(w_i)$.

The actual re-computation of bounds is done in a lazy fashion as follows. In iteration k of the branch-and-bound algorithm, we compute the path P from w_1 to w_k in T' using the parent pointers. Next, we traverse P from w_1 to w_k, and keep track of the final basis in each node using the differences, and of the best available bounds on each variable using the improved bounds that are stored in the nodes on P. Consider some node w_i in this traversal. If the value of z^* that is stored in w_i is less than the actual value, we re-compute the bounds in w_i. If any of the bounds stored in node w_i contradicts with bounds stored in a node w_j that preceded w_i in the traversal, we have found a proof that $X^k = \varnothing$ and we fathom node w_k. If any of the bounds stored in node w_i is implied by a bound stored in a node w_j that preceded w_i in the traversion, we remove it from node w_i.

Consider an execution of the branch-and-bound algorithm and let η denote the number of times we improve on the primal bound. For $w_i \in T'$ let J' denote the non-basic variables in the final basis of node w_i. Assuming that $n \gg m$, the time spent in the re-computation of bounds of node w_i is dominated by the re-computation of the reduced cost from the final basis of node w_i, which is of the order

$$O(\eta|\text{supp}(A_{J'})|). \tag{3.4}$$

In a typical execution of the branch-and-bound algorithm, we improve on the value of z^* only a few times. Moreover, in our applications we use branch-and-cut and branch-and-price algorithms that call sophisticated and often time-consuming subroutines in each iteration of the branch-and-bound algorithm. These observations imply that the bound (3.4) is dominated by the running time of the other computations performed in iteration i of the branch-and-bound algorithm in our applications. We believe that the benefit of having strengthened formulations is worth the extra terms (3.4) in the running time of the branch-and-bound algorithm, as the improved formulations help in reducing the size of the branch-and-bound tree.

3.2.3 GUB Constraints and Tightening Variable Bounds

Assume that row i of the constraints (3.1b) is of the form

$$\boldsymbol{x}(J_i) = 1,$$

where the variables $\boldsymbol{x}_{J_i}$ are required to be non-negative and integer. In this case row i is called a *generalised upper bound (GUB) constraint*. A GUB constraint models the situation in which we have to choose one option from a set of mutually exclusive options. GUB constraints were introduced by Beale and Tomlin [17]. In any feasible integer solution exactly one $j \in J_i$ has $x_j = 1$.

Whenever we have a formulation in which some of the constraints are GUB constraints, we can exploit this by strengthening the bounds on those variables that are in a GUB constraint using a slightly stronger argument than the one presented in Section 3.2.2 (see also Strijk, Verweij, and Aardal [107]). Let $I' \subseteq \{1, \ldots, m\}$ be the set of row indices corresponding to GUB constraints. The *conflict graph* induced by the GUB constraints is the graph $G = (V, E)$, where the node set V contains all variable indices that are involved in one or more GUB constraints and the edge set E contains all pairs of variable indices that are involved in at least one common GUB constraint, i.e.,

$$V = \bigcup_{i \in I'} J_i, \text{ and}$$
$$E = \{\{j, k\} \subseteq V \mid j, k \in J_i, i \in I'\}.$$

For a variable index $j \in V$, we denote by $N(j)$ the set of variable indices k that are adjacent to j in G, i.e., $N(j) = \{k \mid \{j, k\} \in E\}$.

The stronger arguments can be applied to all variables that are involved in at least one GUB constraint. For $j \in V$, whenever x_j has value one, the GUB constraints imply that $\boldsymbol{x}_{N(j)} = \mathbf{0}$, and whenever x_j has value zero the GUB constraints imply that for at least one $k \in N(j)$ x_k has value one. The strengthened argument for modifying the upper bound on x_j takes into account the reduced cost of the variables $\boldsymbol{x}_{N(j)}$, and the strengthened argument for modifying the lower bound on x_j takes into account the reduced cost of the variables x_k and $\boldsymbol{x}_{N(k)}$ for some properly chosen $k \in N(j)$.

Let z^* again denote the value of the best known solution in X. Consider iteration i in which (3.3) is feasible and let $(\boldsymbol{x}^{\mathrm{LP}}, \boldsymbol{\pi})$ be an optimal primal-dual pair to (3.3) in iteration i where $\boldsymbol{\pi} = (\boldsymbol{c}_B^T A_B^{-1})^T$ for some basis $B \subseteq \{1, \ldots, n\}$. Let $z^{\mathrm{LP}} = z(\boldsymbol{x}^{LP})$, and let $L, U \subseteq \{1, \ldots, n\}$ be the maximal sets of variable indices with $\boldsymbol{c}_L^{\boldsymbol{\pi}} < \mathbf{0}$ and $\boldsymbol{c}_U^{\boldsymbol{\pi}} > \mathbf{0}$. The strengthened bounds $\tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i \in \{0, 1\}^V$ are defined as

$$\tilde{l}_j^i = \begin{cases} \max(0, 1 + \lceil (z^* - z^{\mathrm{LP}}) / \min\{-\tilde{c}_k^{\boldsymbol{\pi}} \mid k \in N(j)\} \rceil), & \text{if } j \in U, \\ 0, & \text{otherwise,} \end{cases}$$

and

$$\tilde{u}_j^i = \begin{cases} \min(1, \lfloor (z^* - z^{\mathrm{LP}}) / \tilde{c}_j^{\boldsymbol{\pi}} \rfloor), & \text{if } j \in L, \\ 1, & \text{otherwise,} \end{cases}$$

where for each $j \in V \setminus U$

$$\tilde{c}_j^{\boldsymbol{\pi}} = c_j^{\boldsymbol{\pi}} - \boldsymbol{c}^{\boldsymbol{\pi}}(N(j) \cap U).$$

Proposition 3.2. *All $\boldsymbol{x}^{IP} \in X^i$ with $z(\boldsymbol{x}^{IP}) \geq z^*$ satisfy $\tilde{\boldsymbol{l}}^i \leq \boldsymbol{x}_V^{IP} \leq \tilde{\boldsymbol{u}}^i$.*

Proof. Let $\boldsymbol{x}^{\mathrm{LP}}, \boldsymbol{\pi}, z^{\mathrm{LP}}, L, U$ be as in the construction of $z^*, i, \tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i$. Assume that there exists a vector $\boldsymbol{x}^{\mathrm{IP}} \in X^i$ with $z(\boldsymbol{x}^{\mathrm{IP}}) \geq z^*$. As in Proposition 3.1 we

can write

$$z(\boldsymbol{x}^{\mathrm{IP}}) = \boldsymbol{\pi}^T\boldsymbol{b} + (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T\boldsymbol{x}_L^{\mathrm{IP}} + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T\boldsymbol{x}_U^{\mathrm{IP}}, \text{ and}$$
$$z(\boldsymbol{x}^{\mathrm{LP}}) = \boldsymbol{\pi}^T\boldsymbol{b} + (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T\boldsymbol{l}_L^i + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T\boldsymbol{u}_U^i.$$

Observe that $\boldsymbol{x}^{\mathrm{IP}} \in P_i$ implies $\boldsymbol{x}_L^{\mathrm{IP}} \geq \mathbf{0}$ and $\boldsymbol{x}_U^{\mathrm{IP}} \leq \mathbf{1}$, so $\boldsymbol{x}_U^{\mathrm{IP}} - \mathbf{1} \leq \mathbf{0}$. Now, choose $j \in L \cap V$ arbitrarily. Either $x_j^{\mathrm{IP}} = 0$ or $x_j^{\mathrm{IP}} = 1$. In the case that $x_j^{\mathrm{IP}} = 0$ we directly have $x_j^{\mathrm{IP}} \leq \tilde{u}_j^i$ since $\tilde{u}_j^i \geq 0$. So assume that $x_j^{\mathrm{IP}} = 1$. It follows that $\boldsymbol{x}_{N(j)}^{\mathrm{IP}} = \mathbf{0}$. But then,

$$\begin{aligned} z^* - z^{\mathrm{LP}} &\leq z(\boldsymbol{x}^{\mathrm{IP}}) - z(\boldsymbol{x}^{\mathrm{LP}}) \\ &= (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T(\boldsymbol{x}_L^{\mathrm{IP}} - \boldsymbol{l}_L^i) + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T(\boldsymbol{x}_U^{\mathrm{IP}} - \boldsymbol{u}_U^i) \\ &\leq (\boldsymbol{c}_{L\cap V}^{\boldsymbol{\pi}})^T\boldsymbol{x}_{L\cap V}^{\mathrm{IP}} + (\boldsymbol{c}_{U\cap V}^{\boldsymbol{\pi}})^T(\boldsymbol{x}_{U\cap V}^{\mathrm{IP}} - \mathbf{1}) \\ &\leq c_j^{\boldsymbol{\pi}} x_j^{\mathrm{IP}} + (\boldsymbol{c}_{U\cap N(j)}^{\boldsymbol{\pi}})^T(\boldsymbol{x}_{U\cap N(j)}^{\mathrm{IP}} - \mathbf{1}) \\ &= c_j^{\boldsymbol{\pi}} - \boldsymbol{c}^{\boldsymbol{\pi}}(U \cap N(j)). \end{aligned}$$

Since $c_j^{\boldsymbol{\pi}} - \boldsymbol{c}^{\boldsymbol{\pi}}(U \cap N(j)) < 0$, we find

$$(z^* - z^{\mathrm{LP}})/(c_j^{\boldsymbol{\pi}} - \boldsymbol{c}^{\boldsymbol{\pi}}(U \cap N(j))) \geq 1.$$

Hence $x_j^{\mathrm{IP}} \leq \tilde{u}_j^i$. Because j was chosen arbitrarily we have $\boldsymbol{x}_V^{\mathrm{IP}} \leq \tilde{\boldsymbol{u}}^i$.

The proof that $\boldsymbol{x}_V^{\mathrm{IP}} \geq \tilde{\boldsymbol{l}}^i$ is derived similarly starting from an arbitrarily chosen index $j \in U \cap V$, assuming that $x_j^{\mathrm{IP}} = 0$, and using the observation that $x_j^{\mathrm{IP}} = 0$ implies $x_k^{\mathrm{IP}} = 1$ for some index $k \in N(j)$, which in turn implies that $\boldsymbol{x}_{N(k)}^{\mathrm{IP}} = \mathbf{0}$. □

The strengthened criteria for setting variables based on reduced cost can be taken into account in an implementation that stores the reduced cost of the variables in an array by replacing $c_j^{\boldsymbol{\pi}}$ by $\min\{-\tilde{c}_k^{\boldsymbol{\pi}} \mid k \in N(j)\}$ for all $j \in U$ and by $\tilde{c}_j^{\boldsymbol{\pi}}$ for all $j \in L$ in this array. Having done this the strengthened bounds can be computed as in Section 3.2.2. The extra time needed for pre-processing the array is $O(|E|)$.

3.2.4 Branching Schemes

Here we describe branching schemes for LP-based branch-and-bound. After we have solved the LP relaxation (3.3) in iteration k, and we have found an optimal solution $\boldsymbol{x}^k \in P^k$ but $\boldsymbol{x}^k \notin X^k$, we have to replace P^k by two sets P_1^k and P_2^k.

Branching on Variables

The most common branching scheme in LP-based branch-and-bound is to select $j \in J$ such that x_j^k is fractional and to define P_1^k and P_2^k as

$$P_1^k = P^k \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid x_j \leq \lfloor x_j^k \rfloor\}, \text{ and } P_2^k = P^k \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid x_j \geq \lceil x_j^k \rceil\}.$$

This scheme was first proposed by Dakin [32] and is called *variable branching*. Its correctness follows from the observation that every solution $\boldsymbol{x} \in X$ satisfies $x_j \leq \lfloor x_j^k \rfloor$ or $x_j \geq \lceil x_j^k \rceil$.

The choice of the index j can be made in different ways. One way is to make a decision based on $\boldsymbol{x}^k$, possibly in combination with $\boldsymbol{c}$ and $\boldsymbol{x}^*$. An example of such a scheme, which we will refer to as *Padberg-Rinaldi branching* because of its strong resemblance with the branching rule proposed by Padberg and Rinaldi [92], is as follows. The goal here is to find an index j such that the fractional part of x_j^k is close to $\frac{1}{2}$ and $|c_j|$ is large. For each $j \in J$, let $f_j = x_j^k - \lfloor x_j^k \rfloor$ be the fractional part of x_j^k. First, we determine the values $L, U \in [0,1]$ such that

$$L = \max_{j \in J}\{f_j \mid f_j \leq \tfrac{1}{2}\}, \qquad \text{and} \qquad U = \min_{j \in J}\{f_j \mid f_j \geq \tfrac{1}{2}\}.$$

Next, given a parameter $\alpha \in [0,1]$ we determine the set of variable indices

$$J' = \{j \in J \mid (1-\alpha)L \leq f_j \leq U + \alpha(1-U)\}.$$

The set J' contains the variables that are sufficiently close to $\frac{1}{2}$ to be taken into account. From the set J', we choose an index j that maximises $|c_j|$ as the one to define P_1^k and P_2^k as before. In our implementation we have $\alpha = \frac{1}{2}$ as suggested by Padberg and Rinaldi.

Branching on GUB Constraints

Let $\boldsymbol{x}^k \in P^k$ be as above, and let $J^i = \operatorname{supp}(\boldsymbol{a}_i)$ denote the support of row i of the constraint matrix. Suppose that row i is a GUB constraint and that $\boldsymbol{x}^k_{J^i}$ has fractional components. Partition J^i into two non-empty disjoint sets J_1^i, J_2^i. Observe that any $\boldsymbol{x} \in X$ satisfies $\boldsymbol{x}(J_1^i) = 0$ or $\boldsymbol{x}(J_2^i) = 0$. Hence, we can define P_1^k and P_2^k by

$$P_1^k = P^k \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{x}_{J_1^i} \leq \boldsymbol{0}\}, \quad \text{and} \quad P_2^k = P^k \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{x}_{J_2^i} \leq \boldsymbol{0}\}.$$

The branching scheme defined this way is called *GUB branching* and is due to Beale and Tomlin [17]. GUB branching is generally believed to yield a more balanced branch-and-bound tree of smaller height, and hence of smaller size.

Before we can apply GUB branching, we need a way to choose the sets J_1^i and J_2^i. The most common way to do this is by using an ordering of the variables in J that is specified as input to the branch-and-bound algorithm. Then J is partitioned in such a way that the elements of J_1^i and J_2^i appear consecutively in the specified order. If we have such an ordering then J is called a *special ordered set* (more details can be found in Nemhauser and Wolsey [88]).

We follow a different approach, which is motivated by the observation that the problems from Part II do not exhibit a special ordering. Our aim is to choose J_1^i and J_2^i such that $\boldsymbol{x}^k(J_1^i)$ and $\boldsymbol{x}^k(J_2^i)$ are close to $\frac{1}{2}$. We do this by considering the problem

$$\max_S\{\boldsymbol{x}^k(S) | S \subseteq J, \boldsymbol{x}^k(S) \leq \tfrac{1}{2}\}, \tag{3.5}$$

which is an instance of the subset-sum problem. A strongly polynomial time approximation scheme for the subset-sum problem was given by Ibarra and Kim [61]. We use their algorithm to find J_1^i such that $(1 - \frac{1}{1000})\boldsymbol{x}^k(S^*) \leq \boldsymbol{x}^k(J_1^i) \leq \frac{1}{2}$, where S^* denotes an optimal solution to (3.5), and take $J_2^i = J \setminus J_1^i$. Note that the sets J_1^i, J_2^i constructed this way will be non-empty if $\boldsymbol{x}_J^k$ has fractional components. The precision of $\frac{1}{1000}$ in the calculation of J_1^i is arbitrary but seems to work well in our implementation.

3.2.5 Branching Decisions using Pseudo-Cost

Now that we have seen different branching schemes, we discuss how to choose between the options, that is, how to compare different branching decisions with each other and how to decide on which one to apply. We do this by using *degradation estimates*, that is, estimates on the degradation of the objective function that are caused by enforcing the branching decision. Degradation estimates have been studied by Driebeek [43], Tomlin [111], Bénichou, Gauthier, Girodet, Hentges, Ribière, and Vincent [18], Gauthier and Ribière [49], and most recently by Linderoth and Savelsbergh [79].

Focus on iteration k of the branch-and-bound algorithm and let $\boldsymbol{x}^k \in P^k$ be as in Section 3.2.4. Observe that all branching decisions of Section 3.2.4 are of the form

$$P_1^k = P^k \cap D_1, \qquad \text{and} \qquad P_2^k = P^k \cap D_2,$$

where $D_b \subseteq \mathbb{R}^n$ enforces the altered variable bounds for each $b \in \{1, 2\}$. In the following, we call D_b a *branch* ($b \in \{1, 2\}$), and a pair (D_1, D_2) a *branching option*. Our goal is to decide which branching option to apply. Once we decided to branch using a pair of branches (D_1, D_2) we will refer to it as a *branching decision*.

So we have a fractional $\boldsymbol{x}^k$, and after applying the ideas of Section 3.2.4 we find ourselves with a set of branching options from which we have to choose the one we will enforce. Denote the set of branching options by $\mathcal{D}$, and let $\mathcal{D} = \{(D_1^1, D_2^1), \ldots, (D_1^N, D_2^N)\}$. Later in this section we will discuss in detail how to obtain the set $\mathcal{D}$. We may assume that $\boldsymbol{x}^k \notin D_b^i$ for all choices of (i, b). For each $D = D_b^i$ a measure of the distance from $\boldsymbol{x}^k$ to D can be defined as

$$f(D) = \begin{cases} x_j^k - \tilde{u}_j, & \text{if } D = \{\boldsymbol{x} \in \mathbb{R}^n \mid x_j \leq \tilde{u}_j\}, \\ \tilde{l}_j - x_j^k, & \text{if } D = \{\boldsymbol{x} \in \mathbb{R}^n \mid x_j \geq \tilde{l}_j\}, \\ \boldsymbol{x}^k(J'), & \text{if } D = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{x}_{J'} \leq \boldsymbol{0}\}. \end{cases}$$

Note that $0 < f(D_b^i) < 1$ for all $(i, b) \in \{1, \ldots, N\} \times \{1, 2\}$ if we select all branching options as in Section 3.2.4. The per-unit degradation of the objective function caused by enforcing D_b^i is called the *pseudo-cost* of D_b^i, and is given by

$$p_b^i = \frac{\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in P^k\} - \max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in P^k \cap D_b^i\}}{f(D_b^i)}$$

for each $(i, b) \in \{1, \ldots, N\} \times \{1, 2\}$. Actually computing the pseudo-cost for all branches is time consuming, and is not believed to result in improved running times.

It is reported by Gauthier and Ribiére, and more recently by Linderoth and Savelsbergh that, although the pseudo-cost of each variable are different in each iteration of the branch-and-bound algorithm, the order of magnitude of the pseudo-cost of each variable is the same in all but a few iterations. Therefore, instead of calculating the true pseudo-costs, we maintain estimates $\tilde{p}_b^i$ of p_b^i for all D_b^i that have been of interest during the execution of the branch-and-bound algorithm. For each variable x_j with $j \in J$ we store the values of the estimates $\tilde{p}_1^i$ and $\tilde{p}_2^i$ of the degradation of the branches $\{\boldsymbol{x} \in \mathbb{R}^n \mid x_j \leq \tilde{u}_j\}$ and $\{\boldsymbol{x} \in \mathbb{R}^n \mid x_j \geq \tilde{l}_j\}$. For each $J' \subseteq J$ that was obtained by partitioning the index set of a GUB constraint we store the value of the estimate $\tilde{p}_b^i$ corresponding to the branch $D_b^i = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{x}_{J'} \leq \boldsymbol{0}\}$ in a hash table $\mathcal{H} : 2^J \to \mathbb{R}$ using open addressing [29].

We follow the ideas of Linderoth and Savelsbergh in that we do allow our algorithm to spend some time to compute good initial estimates. These initial estimates are calculated using the dual simplex algorithm with an iteration limit $L(d)$ that is a function of the depth d of node v_k in the branch-and-bound tree, i.e.,

$$L(d) = \max\left\{\left\lceil \frac{T\gamma}{2^d N} \right\rceil, 3\right\},$$

where T denotes the maximum amount of time we are willing to spend on level d of the tree (in seconds), and γ is an estimate of the number of simplex iterations performed per second. In our implementation T is four minutes and γ is determined after solving the LP in the root node. We maintain the degradation estimates during the execution of the algorithm by letting $\tilde{p}_b^i$ be the average over all observed degradations caused by enforcing D_b^i that led to feasible relaxations (discarding the initial estimate). The average degradation of a branch D can be maintained in $O(1 + T_D)$ time for each time that we branched on D, where T_D indicates the time needed to access the stored pseudo-cost of D. If D is a variable branch then $T_D = O(1)$, and if D is a GUB branch that affects $\boldsymbol{u}_{J'}$ for some set $J' \subseteq J$ then T_D is a random variable and $E[T_D] = O(|J'|)$.

The actual branching decision is made as follows. For each D_b^i we estimate a degradation

$$d_b^i = \tilde{p}_b^i f(D_b^i).$$

We choose to branch on $(D_1^{i^*}, D_2^{i^*}) \in \mathcal{D}$ determined by

$$i^* = \arg \max_{i \in \{1,\ldots,N\}} \{\alpha_1 \min(d_1^i, d_2^i) + \alpha_2 \max(d_1^i, d_2^i)\},$$

where in our implementation (α_1, α_2) equals $(2, 1)$ as suggested by Linderoth and Savelsbergh. We have only one exception to this rule, namely, if $d_1^{i^*} = d_2^{i^*} = 0$ then we use the Padberg-Rinaldi scheme. In this case all pseudo-costs are zero, and hence provide a poor criterion for comparing branching options.

To complete the description of our branching scheme, we have to specify how we obtain the set of branching options $\mathcal{D}$. We do this by including in $\mathcal{D}$ all branching options that are derived from fractional variables x_j with $j \in J$. In addition, we add to $\mathcal{D}$ a selected number of branching options that are derived from GUB constraints. One might be tempted to include in $\mathcal{D}$ a branching option derived from each GUB constraint that has fractional variables. However, if the number of GUB constraints in the formulation is relatively large then such a scheme would waste too much time in selecting GUB constraints for branching, eliminating the potential benefit obtained from branch selection by pseudo-cost.

Instead we use the following approach, which takes into account that GUB constraints may be added in later stages of the algorithm (this can occur in a branch-and-cut algorithm). Let $I_1 \subseteq \{1, \ldots, m\}$ denote the set of row-indices of GUB constraints, so for each $i \in I_1$ row i has the form

$$\boldsymbol{x}(J^i) = 1,$$

$\boldsymbol{x}_{J^i} \geq \mathbf{0}$, and $J^i \subseteq J$. For each $i \in I_1$ we keep track of the average pseudo-cost p_b^i of branches D_b^i obtained by partitioning J^i, that we use as a heuristic forecast of the pseudo-cost of the GUB branches obtained from row i in this iteration of the branch-and-bound algorithm (although the partitioning of J^i, and hence the resulting branching decision, might be totally different). We pre-select all GUB constraints that have at least four fractional variables in their support. Let $I_2 \subseteq I_1$ be the set of row indices corresponding to GUB constraints such that $\boldsymbol{x}_{J^i}$ has at least four fractional components for all $i \in I_2$. Let $I_3 \subseteq I_2$ be the set of row indices corresponding to GUB constraints from which we did derive a branching option, and let $I_4 = I_2 \setminus I_3$ be the set of row indices corresponding to GUB constraints from which we did not derive a branching option yet. For the rows indexed by I_3 we already have a heuristic forecast of the pseudo-cost of the branching decisions obtained from them, and for the rows indexed by I_4 we have not. From I_3, we select the $K(d)$ constraints with highest average pseudo-cost, and add those to I_4. Here, $K(d)$ is again a function of the depth d of node v^k in the branch-and-bound tree, and is given by

$$K(d) = \max\left\{\left\lceil \frac{m}{2^{d-1}} \right\rceil, 10\right\}.$$

For each $i \in I_4$ we partition J^i as in the normal GUB branching procedure and add the resulting pair of branches to $\mathcal{D}$ if the partition results in two sets J_1^i, J_2^i that each contain at least two fractional variables.

3.2.6 Logical Implications

Whenever we tighten variable bounds either by branching on them or by using reduced cost criteria, we try to tighten more bounds by searching for *logical implications* of the improved bounds. This is done using the structure of the problem at hand, so we do not go into detail here.

Consider iteration k of the branch-and-bound algorithm and let $\boldsymbol{x}^k \in P^k$ be as before. After enforcing the improved bounds obtained by calculating logical

implications we obtain a relaxation

$$\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in \tilde{P}^k\}$$

for some $\tilde{P}^k \subset P^k$. It may occur that $\boldsymbol{x}^k \notin \tilde{P}^k$. If this is the case then we iterate by resolving the strengthened LP, and proceed again with the strengthening of variable bounds.

3.3 Branch-and-Cut

In this section we refine the LP-based branch-and-bound algorithm from Section 3.2 in such a way that it is no longer necessary to include all constraints in the LP formulation that is solved. The resulting refinement is known as a *branch-and-cut* algorithm, and allows us to use formulations with a large or possibly exponential number of constraints. Formulations with an exponential number of constraints are of interest because for some problems it is the only way to formulate them, and for other problems such formulations are significantly stronger than formulations of polynomial size. In both cases we need a *cutting plane* algorithm to solve the linear program, explained in Section 3.3.1, that iteratively adds violated *valid inequalities* to the LP formulation. The cutting plane algorithm relies on problem-specific subroutines that are known as *separation algorithms*. Some implementational issues that arise when dynamically adding constraints are discussed in Section 3.3.2. In Sections 3.3.3–3.3.5 we review some known classes of valid inequalities together with their separation algorithms, which we will use in Part II.

3.3.1 Valid Inequalities and The Cutting Plane Algorithm

Let $P \subseteq \mathbb{R}^n$, and $(\boldsymbol{\pi}, \pi_0) \in \mathbb{R}^n \times \mathbb{R}$. The inequality $\boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0$ is *valid* for P if

$$P \subseteq \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0\}.$$

Now, let P be a polyhedron in $\mathbb{R}^n$ and let $\boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0$ be a valid inequality for P. The set

$$F = P \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{\pi}^T\boldsymbol{x} = \pi_0\}$$

is called a *face* of P. The valid inequality $\boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0$ is said to *define* F. We say that F is a *proper* face of P if $F \neq \varnothing$ and $F \neq P$. A proper face is called a *facet* if it is not contained in any other proper face of P. Facet-defining valid inequalities are the strongest among all valid inequalities as they cannot be redundant (see e.g. Wolsey [125]). Recent surveys devoted to the subject of valid inequalities are by Aardal and van Hoesel [1] and by Marchand, Martin, Weismantel and Wolsey [82].

Recall the formulation of the mixed integer programming problem from Section 3.1. Let $\Pi \subseteq \mathbb{R}^n \times \mathbb{R}$ be such that each $(\boldsymbol{\pi}, \pi_0) \in \Pi$ defines a valid inequality for $\text{conv}(X)$. Then, the linear program

$$\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in P,\ \boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0 \text{ for all } (\boldsymbol{\pi}, \pi_0) \in \Pi\} \tag{3.6}$$

```
cuttingPlaneSolver(z, A, b, l, u, separation, α1, α2)
{
    Π̄ := ∅; i := 0;
    do { i := i + 1;
        solve LP max{z(x) | Ax = b, l ≤ x ≤ u, π^T x ≤ π0 ∀(π, π0) ∈ Π̄};
        if LP infeasible return infeasible;
        let x* be an optimal solution of LP; z^i := z(x*);
        improve bounds and calculate logical implications, thus refining LP;
        if x* violates new bounds {
            resolve LP;
            if LP infeasible return infeasible;
            let x* be an optimal solution; z^i := z(x*);
        }
        Π' := separation(x*); Π̄ := Π̄ ∪ Π';
        ξ := max_{(π,π0)∈Π'} π0 − π^T x*;
    } while (i > 1 implies z^{i−1} − z^i ≥ α1) and ξ ≥ α2 and Π' ≠ ∅;
    return x*;
}
```

Algorithm 3.1: The Cutting Plane Algorithm

is a relaxation of the mixed integer programming problem (3.1). Hence, if we can solve (3.6) efficiently, then we can incorporate it in an LP-based branch-and-bound algorithm.

The *cutting plane algorithm* to solve (3.6) works in iterations (starting from 1), and maintains a set $\bar{\Pi} \subseteq \Pi$. Initially, the set $\bar{\Pi}$ is empty. In iteration i, we solve the linear program

$$\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in P, \boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0 \text{ for all } (\boldsymbol{\pi}, \pi_0) \in \bar{\Pi}\}. \tag{3.7}$$

If this problem is infeasible then the cutting plane algorithm reports so and terminates. Otherwise, let $\boldsymbol{x}^*$ denote the optimal solution to (3.7), and let z^i denote the value of $z(\boldsymbol{x}^*)$. We first try to improve the bounds using Proposition 3.1. If this succeeds we try to improve more bounds using logical implications as described in Section 3.2.6. If any of these implications are not satisfied by $\boldsymbol{x}^*$, we resolve the LP. If the LP becomes infeasible, then we report so and terminate. Otherwise we update z^i and $\boldsymbol{x}^*$. The cutting plane algorithm then calls the separation algorithm with $\boldsymbol{x}^*$ as input, which returns a set $\Pi' \subseteq \Pi$ such that for each $(\boldsymbol{\pi}, \pi_0) \in \Pi'$ the valid inequality $\boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0$ is violated by $\boldsymbol{x}^*$. Let

$$\xi = \max_{(\boldsymbol{\pi},\pi_0)\in\Pi'} \pi_0 - \boldsymbol{\pi}^T\boldsymbol{x}^*$$

denote the maximum violation of any valid inequality defined by a pair $(\boldsymbol{\pi}, \pi_0) \in \Pi'$. If Π' is empty, or if $i > 1$, $z^{i-1} - z^i < \alpha_1$ and $\xi < \alpha_2$, then the cutting plane algorithm terminates and reports $\boldsymbol{x}^*$ as solution. Otherwise, it adds Π'

to $\bar{\Pi}$ and proceeds with the next iteration. The parameters (α_1, α_2) are used to avoid a phenomenon that is called *tailing-off* (see e.g. Padberg and Rinaldi [92] and Jünger *et al.* [67]) of the cutting plane algorithm. Tailing-off occurs when the separation routines find cuts that do not add much to the strength of the formulation. If this is the case we are basically wasting precious CPU time. In our implementation we have $(\alpha_1, \alpha_2) = (\frac{1}{100}, \frac{1}{10})$. Pseudo-code for the cutting plane algorithm can be found in Algorithm 3.1.

3.3.2 Initialising LP Relaxations and Constraint Pools

When one uses a cutting plane algorithm to solve the LP relaxation in each iteration, the question arises what constraints to keep in the LP formulation that is maintained by the branch-and-bound algorithm. One possible answer would be to keep all constraints that are generated by the separation routines in the formulation. However, this would result in an LP formulation that keeps growing over the execution of the branch-and-cut algorithm. Therefore the linear programs would take longer and longer to solve, which is undesirable.. Keeping all constraints in the formulation is not what we will do.

Focus on any iteration $i > 1$, and let v_j be the parent of node v_i in the branch-and-bound tree. Denote the optimal solution to LP^j by $\boldsymbol{x}^j$. Padberg and Rinaldi [92] initialise the formulation in each iteration $i > 1$ with the constraints of the matrix A, and the constraints generated by the separation routines that were satisfied with equality by $\boldsymbol{x}^j$, This is the approach we follow. Note that all constraints with a non-basic slack variable are satisfied with equality. Initialising the LP formulation associated with iteration i is then accomplished by eliminating valid inequalities from LP^{i-1} that are not satisfied with equality by $\boldsymbol{x}^j$, and adding those valid inequalities present in LP^j that are satisfied with equality and have a non-basic slack variable, next to imposing the bounds given in Section 3.2.1. The information about the status of the slack variables of valid inequalities that are added to the LP formulation is handled in the same way as the information about optimal the basis in Section 3.2.2.

A second issue that arises when one uses the cutting plane algorithm and one initialises the LP formulations as above is the following. It might be advantageous to cache constraints that are produced by the separation algorithm by storing them in a constraint pool, and then checking the constraint pool for violated valid inequalities before calling the separation algorithm. Checking the constraint pool for violated valid inequalities is called *pool separation* and is implemented by computing an inner product $\boldsymbol{\pi}^T\boldsymbol{x}$ for each valid inequality $(\boldsymbol{\pi}, \pi_0)$ in the constraint pool and comparing the result with π_0. If a constraint in the pool is violated in some iteration of the branch-and-cut algorithm, it is reported as violated and not checked again for the remainder of this iteration. As separation algorithms often solve non-trivial combinatorial optimisation problems, this can indeed be advantageous. However, the size of a constraint pool typically grows over the execution of the algorithm, causing the pool separation to take longer and longer as the branch-and-cut algorithm proceeds.

Consider any iteration of the branch-and-cut algorithm and denote the set

of valid inequalities in the constraint pool by

$$\bar{\Pi} = \{(\boldsymbol{\pi}^i)^T \boldsymbol{x} \leq \pi_0^i \mid i \in \{1, \ldots, |\bar{\Pi}|\}\}.$$

To avoid the pool separation from stalling our branch-and-cut algorithm, we delete constraints from the constraint pool at certain times. For $i \in \{1, \ldots, |\bar{\Pi}|\}$ we keep track of the number of iterations of the branch-and-bound algorithm k_1^i that the inequality $(\boldsymbol{\pi}^i)^T \boldsymbol{x} \leq \pi_0^i$ was in the pool since it was last inserted, and the number of iterations k_2^i in which it was actually reported as violated during this period of time. The valid inequality $(\boldsymbol{\pi}^i)^T \boldsymbol{x} \leq \pi_0^i$ is kept in the pool at least as long as $k_1^i \leq \alpha_1$ for some parameter α_1. However, as soon as $k_1^i > \alpha_1$ and $k_2^i / k_1^i < \alpha_2$ we delete it. The advantage of this scheme is that the number of iterations that a constraint stays in the pool is linear in the number of iterations in which it is actually reported as violated. In our implementation we have $(\alpha_1, \alpha_2) = (32, \frac{1}{5})$. In this way a constraint is kept for at least 32 iterations and dropped as soon as it was not violated in at least 20% of the iterations that it was in the pool, which appears to be a reasonable choice of parameters.

3.3.3 Cover Inequalities for Knapsack Sets

Suppose we are given a set of items N, a vector $\boldsymbol{a} \in \mathbb{N}^N$ that gives the size of each item in N, and a knapsack of size $b \in \mathbb{N}$. Let

$$P = \{\boldsymbol{x} \in [0,1]^N \mid \boldsymbol{a}^T \boldsymbol{x} \leq b\}.$$

The set

$$X = P \cap \{0,1\}^N$$

contains all incidence vectors of subsets of N that fit in the knapsack and is called a *knapsack set*. Knapsack sets are of interest to us because they appear as a relaxation of the problem we discuss in Chapter 6. The set $\text{conv}(X)$ is the *knapsack polytope*, which has been studied since the mid-seventies by Padberg [90], Balas [8], Wolsey [122], Hammer, Johnson, and Peled [58], Crowder, Johnson and Padberg [31], and Weismantel [120]. Here we follow the exposition of Wolsey [125].

A *cover* is a set $C \subseteq N$ such that $\boldsymbol{a}(C) > b$. Let C be a cover. A *cover inequality* is an inequality of the form

$$\boldsymbol{x}(C) \leq |C| - 1, \tag{3.8}$$

Cover inequalities are valid inequalities for the knapsack polytope. We say that C is *minimal* if every proper subset $C' \subset C$ has $\boldsymbol{a}(C') \leq b$. A minimal cover inequality is facet-defining if $C = N$.

A Greedy Heuristic for Searching Cover Inequalities

Suppose we are given a fractional solution $\boldsymbol{x}^* \in P$. To find a cover inequality we follow the approach presented by Crowder, Johnson and Padberg [31]. Let

$B \subseteq N$ be the fractional support of $\boldsymbol{x}^*$, and let $U \subseteq N \setminus B$ be the maximal set of items with $\boldsymbol{x}^*_U = \mathbf{1}$. To separate a violated cover inequality, we focus on the set of items B with respect to a knapsack of size $b' = b - \boldsymbol{a}(U)$. Note that the cover inequality (3.8) is equivalent to

$$|C| - \boldsymbol{x}(C) \geq 1.$$

Hence, there exists a cover $C \subseteq B$ with respect to a knapsack of size b' that induces a violated cover inequality in $\boldsymbol{x}^*$ if and only if there exists a set $C \subseteq B$ with $\boldsymbol{a}(C) > b'$ that satisfies

$$|C| - \boldsymbol{x}^*(C) < 1,$$

which is the case if

$$C^* = \arg\min_{C \subseteq B} \left\{ \textstyle\sum_{i \in C}(1 - x^*_i) \mid \boldsymbol{a}(C) > b' \right\}$$

satisfies $|C^*| - \boldsymbol{x}^*(C^*) < 1$.

The cover is computed using a greedy heuristic that first sorts the items in B in non-decreasing order of $(1 - x^*_i)/a_i$. Let $B = \{i_1, \ldots, i_{|B|}\}$ be the resulting ordering. The greedy algorithm proceeds with two stages. In the first stage it finds a cover, and in the second stage it turns it into a minimal cover. Finding a cover is done by determining the smallest index k such that the items $\{i_1, \ldots, i_k\}$ are a cover with respect to a knapsack of size b', i.e., such that $\boldsymbol{a}(\{i_1, \ldots, i_k\}) > b'$. Let k be this smallest index, and let $C = \{i_1, \ldots, i_k\}$. The second stage works in iterations, numbered $j-1, \ldots, 1$. In iteration l, we check whether $C \setminus \{i_l\}$ has $\boldsymbol{a}(C \setminus \{i_l\}) > b'$ and if so, we replace C by $C \setminus \{i_l\}$. Upon termination of the second stage the set C is a minimal cover, and the heuristic returns it. The heuristic can be implemented in $O(|B| \log |B|)$ time.

The cover C that is returned by the heuristic need not be violated. Whether it is violated or not, we will try to strengthen it using the lifting algorithm that is the subject of the next section. If a cover inequality is violated after lifting, we report it.

3.3.4 Lifted Cover Inequalities for Knapsack Sets

Padberg [90], and Wolsey [122, 123] studied *sequential lifting* of valid inequalities. Sequential lifting of valid inequalities for zero-one integer programming problems relies on the following theorem (see also Nemhauser and Wolsey [88, Chapter II.2, Propositions 1.1 and 1.2]):

Theorem 3.3. *Let $S \subseteq \{0,1\}^n$. For $k \in \{0,1\}$, let $S^k = S \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid x_1 = k\}$.*

(i) Suppose

$$\sum_{j=2}^{n} \pi_j x_j \leq \pi_0 \tag{3.9}$$

is valid for S^0. *If* $S^1 = \varnothing$, *then* $x_1 \leq 0$ *is valid for* S. *If* $S^1 \neq \varnothing$, *then*

$$\sum_{j=1}^{n} \pi_j x_j \leq \pi_0 \tag{3.10}$$

is valid for S *for any* $\pi_1 \leq \pi_0 - \max_{\boldsymbol{x} \in S^1} \{\sum_{j=2}^{n} \pi_j x_j\}$. *If (3.9) defines a face of* $\operatorname{conv}(S^0)$ *of dimension* k, *and* π_1 *is chosen maximal, then (3.10) defines a face of* $\operatorname{conv}(S)$ *of dimension* $k+1$.

(ii) Suppose that (3.9) is valid for S^1. *If* $S^0 = \varnothing$, *then* $x_1 \geq 1$ *is valid for* S. *If* $S^0 \neq \varnothing$, *then*

$$\sum_{j=1}^{n} \pi_j x_j \leq \pi_0 + \pi_1 \tag{3.11}$$

is valid for S *for any* $\pi_1 \geq \max_{\boldsymbol{x} \in S^0} \{\sum_{j=2}^{n} \pi_j x_j\} - \pi_0$. *If (3.9) defines a face of* $\operatorname{conv}(S^1)$ *of dimension* k, *and* π_1 *is chosen minimal, then (3.11) defines a face of* $\operatorname{conv}(S)$ *of dimension* $k+1$.

Theorem 3.3 is a special case of the theorem given by Wolsey [123, Theorem 1] for general integer programming problems.

Consider again the sets P and X defined by a set of items N, a vector $\boldsymbol{a} \in \mathbb{N}^N$ that gives the size of each item, and a knapsack of size b as in Section 3.3.3. Suppose we are given a fractional solution $\boldsymbol{x}^* \in P$ and that we have found a cover inequality

$$\boldsymbol{x}(C) \leq |C| - 1 \tag{3.12}$$

that is valid for the set

$$\big\{\boldsymbol{x} \in \{0,1\}^N \mid \textstyle\sum_{j \in C} a_j x_j \leq b - \boldsymbol{a}(U)\big\},$$

using the separation algorithm described in Section 3.3.3, where U is again the maximal set of items such that $\boldsymbol{x}^*_U = \mathbf{1}$. Van Roy and Wolsey [113] give an algorithm that iteratively applies Theorem 3.3 starting from (3.12) to obtain a lifted cover inequality of the form

$$\boldsymbol{x}(C) + \sum_{j \in D} \alpha_j x_j \leq |C| - 1 + \sum_{j \in U} \alpha_j (1 - x_j),$$

where C, D and U are mutually disjoint sets of items. In each iteration, one coefficient α_j is computed using a dynamic-programming algorithm for the knapsack problem (see e.g. Nemhauser and Wolsey [88, Chapter II.6, Proposition 1.6]). Gu, Nemhauser and Savelsbergh [57] discuss modern implementations of these techniques.

3.3.5 Mod-k Inequalities for Integer Programming

Suppose we are given natural numbers m, n, a matrix $A \in \mathbb{Z}^{m\times n}$, and a vector $\boldsymbol{b} \in \mathbb{Z}^m$. Let

$$P = \{\boldsymbol{x} \in \mathbb{R}^n \mid A\boldsymbol{x} \leq \boldsymbol{b}\},$$

and let

$$X = P \cap \mathbb{Z}^n.$$

A *Chvátal-Gomory cut* is a valid inequality for $\text{conv}(X)$ of the form

$$\boldsymbol{\lambda}^T A\boldsymbol{x} \leq \lfloor \boldsymbol{\lambda}^T \boldsymbol{b} \rfloor, \tag{3.13}$$

where $\boldsymbol{\lambda} \in \mathbb{R}^m_+$ satisfies $\boldsymbol{\lambda}^T A \in \mathbb{Z}^n$. If $\boldsymbol{\lambda} \in \{0, 1/k, \ldots, (k-1)/k\}^m$ for any $k \geq 2$, then the Chvátal-Gomory cut defined by $\boldsymbol{\lambda}$ is called a *mod-k cut*. Let $\boldsymbol{\lambda} \in \{0, 1/k, \ldots, (k-1)/k\}^m$. Given $\boldsymbol{x}^* \in P$, the violation achieved by $\boldsymbol{x}^*$ of the mod-k cut defined by $\boldsymbol{\lambda}$ is given by

$$\boldsymbol{\lambda}^T A\boldsymbol{x}^* - \lfloor \boldsymbol{\lambda}^T \boldsymbol{b} \rfloor.$$

Since $\boldsymbol{\lambda}^T A\boldsymbol{x} \leq \boldsymbol{\lambda}^T \boldsymbol{b}$ is satisfied by $\boldsymbol{x}^*$, the maximal violation of a mod-k cut that can be obtained by $\boldsymbol{x}^*$ is $(k-1)/k$. Mod-k cuts for which $\boldsymbol{x}^*$ achieves this violation are called *maximally violated* by $\boldsymbol{x}^*$. The separation of mod-2 cuts was studied by Caprara and Fischetti [22]. The separation of mod-k cuts as treated in this section is by Caprara, Fischetti, and Letchford [23, 24]. The study of mod-k cuts is motivated by the fact that several well-known classes of valid inequalities for which no efficient separation algorithms were known, can be interpreted as mod-k cuts. This includes the class of comb inequalities of the travelling salesman problem.

Suppose we are given a fractional solution $\boldsymbol{x}^* \in P$. There exists a mod-k cut that is violated by $\boldsymbol{x}^*$ if and only if there exists $\boldsymbol{\lambda} \in \{0, 1/k, \ldots, (k-1)/k\}^m$ with

$$\boldsymbol{\lambda}^T A\boldsymbol{x}^* - \lfloor \boldsymbol{\lambda}^T \boldsymbol{b} \rfloor > 0 \text{ and } \boldsymbol{\lambda}^T A \in \mathbb{Z}^n,$$

which is the case if and only if

$$\zeta = \max_{\boldsymbol{\lambda}} \left\{ \boldsymbol{\lambda}^T A\boldsymbol{x}^* - \lfloor \boldsymbol{\lambda}^T \boldsymbol{b} \rfloor \mid \boldsymbol{\lambda}^T A \in \mathbb{Z}^n, \boldsymbol{\lambda} \in \{0, 1/k, \ldots, (k-1)/k\}^m \right\} \tag{3.14}$$

satisfies $\zeta > 0$. Let $\boldsymbol{s}^* = \boldsymbol{b} - A\boldsymbol{x}^*$ denote the slack of $\boldsymbol{x}^*$. By substituting $A\boldsymbol{x}^* = \boldsymbol{b} - \boldsymbol{s}^*$, $\boldsymbol{\lambda}^T \boldsymbol{b} - \lfloor \boldsymbol{\lambda}^T \boldsymbol{b} \rfloor = \theta/k$, and $\boldsymbol{\lambda} = \frac{1}{k}\boldsymbol{\mu}$ in (3.14) and observing that $\boldsymbol{\lambda}^T \boldsymbol{b} - \lfloor \boldsymbol{\lambda}^T \boldsymbol{b} \rfloor = \theta/k$ if and only if $\boldsymbol{b}^T \boldsymbol{\mu} \equiv \theta \pmod{k}$ and that $\boldsymbol{\lambda}^T A \in \mathbb{Z}^n$ if and only if $A^T \boldsymbol{\mu} \equiv \boldsymbol{0} \pmod{k}$, we obtain the equivalent formulation

$$\begin{aligned}
\zeta = \frac{1}{k} \max_{(\boldsymbol{\mu},\theta)} \quad & \theta - (\boldsymbol{s}^*)^T \boldsymbol{\mu} && \text{(3.15a)}\\
\text{subject to} \quad & A^T \boldsymbol{\mu} \equiv \boldsymbol{0} \pmod{k}, && \text{(3.15b)}\\
& \boldsymbol{b}^T \boldsymbol{\mu} \equiv \theta \pmod{k}, && \text{(3.15c)}\\
& (\boldsymbol{\mu}, \theta) \in \{0, \ldots, k-1\}^{m+1}. && \text{(3.15d)}
\end{aligned}$$

Caprara *et al.* show that (3.15) is $\mathcal{NP}$-hard using a reduction from the maximum cut problem. Furthermore, they show that the problem is polynomially solvable if we restrict ourselves to maximally violated mod-k inequalities. If $(\boldsymbol{\mu}^*, \theta^*)$ is an optimal solution to (3.15), then $\boldsymbol{\lambda} = \frac{1}{k}\boldsymbol{\mu}^*$ defines a mod-k cut that is violated by $\boldsymbol{x}^*$ if $\zeta > 0$, and ζ is the violation obtained by $\boldsymbol{x}^*$. Let $I = \{i \in \{1, \ldots, m\} \mid s_i^* = 0\}$ denote the set of row indices that have a slack equal to zero with respect to $\boldsymbol{x}^*$. From the objective function of (3.15) it is immediately clear that $(\boldsymbol{\mu}^*, \theta^*)$ defines a maximally violated mod-k cut if and only if $\theta^* = k - 1$ and $\mu_i^* = 0$ for all i with $s_i^* > 0$. From this it follows that a maximally violated mod-k exists if and only if the following system of mod-k congruences has a solution:

$$A_I^T \boldsymbol{\mu} \equiv \mathbf{0} \pmod{k}, \tag{3.16a}$$

$$b_I^T \boldsymbol{\mu} \equiv k - 1 \pmod{k}, \tag{3.16b}$$

$$\boldsymbol{\mu} \in \{0, \ldots, k-1\}^I. \tag{3.16c}$$

Determining feasibility of (3.16) and finding a solution to it if it is feasible can be done in $O(mn \min(m, n))$ time if k is prime using standard Gaussian elimination in $GF(k)$. Moreover, they show that the existence of a maximally violated mod-k cut for any non-prime value of k implies the existence of a maximally violated mod-l cut for every prime factor l of k. Therefore, we can restrict our attention to the separation of maximally violated mod-k cuts for which k is prime.

In our implementation the system of mod-k congruences (3.16) is solved by computing an LU-factorisation (see e.g. Chvátal [28]) of a sub-matrix of (3.16a)–(3.16b) that has full rank using arithmetic in $GF(k)$ (see e.g. Cormen *et al.* [29]), and checking the solution against the remaining constraints. To preserve the sparsity of the matrix we use a greedy heuristic that selects the next pivot from the sparsest remaining non-zero row. Our implementation is preliminary in that there are more sophisticated heuristics to preserve sparsity, and in that we compute the factorisation from scratch over and over again.

3.4 Branch-and-Price

Now that we have seen how to adapt the LP-based branch-and-bound algorithm for integer linear programming formulations with a large number of constraints, we consider how to adapt the LP-based branch-and-bound algorithm in such a way that it is no longer necessary to include all variables in the formulation that is passed to the LP solver. This allows us to use formulations that have a large, or even exponential number of variables. Problems that give rise to such formulations have been studied since the nineteen eighties. These studies include the ones by Desrosiers, Soumis, and Desrochers [40], Desrochers, Desrosiers, and Solomon [38], Barnhart, Johnson, Nemhauser, Savelsbergh, and Vance [14], Vanderbeck and Wolsey [115], and Vanderbeck [114].

3.4.1 LP Column Generation

Suppose we are given a matrix

$$A = \begin{pmatrix} a_{I_0} \\ a_{I \setminus I_0} \end{pmatrix} \in \mathbb{Z}^{m \times n},$$

and a vector

$$\boldsymbol{b} = \begin{pmatrix} \boldsymbol{b}_{I_0} \\ \boldsymbol{b}_{I \setminus I_0} \end{pmatrix} \in \mathbb{Z}^{m},$$

where m, n are natural numbers, and $I_0 \subseteq I = \{1, \dots, m\}$ are sets of row indices. Consider the linear programming problem

$$\max\{z(\boldsymbol{x}) = \boldsymbol{c}^T \boldsymbol{x} \mid a_{I_0} \boldsymbol{x} = \boldsymbol{b}_{I_0}, \boldsymbol{x} \in \operatorname{conv}(X)\}, \tag{3.17}$$

where $\boldsymbol{c} \in \mathbb{Z}^n$ is the cost vector, and X is defined by the sub-matrix $a_{I \setminus I_0}$ of A and bound vectors $\boldsymbol{l}, \boldsymbol{u} \in \mathbb{Z}^n$ as follows:

$$P = \{\boldsymbol{x} \in \mathbb{R}^n \mid a_{I \setminus I_0} \boldsymbol{x} = \boldsymbol{b}_{I \setminus I_0}, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\}$$

and

$$X = P \cap \mathbb{Z}^n.$$

We will reformulate (3.17) using *Dantzig-Wolfe decomposition* [36] to obtain a formulation with a possibly exponential number of variables. Suppose that $a_{I \setminus I_0}$ has a block-diagonal structure, and let $I_1, \dots, I_k \subseteq I \setminus I_0$ and $J_1, \dots, J_k \subseteq \{1, \dots, n\}$ be a partitioning of the row and variable indices of $a_{I \setminus I_0}$, respectively, such that

$$A = \begin{pmatrix} A^1 & A^2 & \dots & A^k \\ A_{I_1 J_1} & & & \\ & A_{I_2 J_2} & & \\ & & \dots & \\ & & & A_{I_k J_k} \end{pmatrix}, \tag{3.18}$$

where $A^i = A_{I_0 J_i}$. For each $i \in \{1, \dots, k\}$, let

$$P_i = \{\boldsymbol{x} \in \mathbb{R}^{J_i} \mid A_{I_i J_i} \boldsymbol{x} = \boldsymbol{b}_{I_i}, \boldsymbol{l}_{J_i} \leq \boldsymbol{x} \leq \boldsymbol{u}_{J_i}\},$$

and

$$X_i = P_i \cap \mathbb{Z}^{J_i}.$$

To simplify the following discussion, assume that X_i is finite for each $i \in \{1, \dots, k\}$.

Note that any fractional solution $\boldsymbol{x} \in \operatorname{conv}(X)$ satisfies $\boldsymbol{x}_{J_i} \in \operatorname{conv}(X_i)$, so $\boldsymbol{x}_{J_i}$ can be expressed as a convex combination of the elements of X_i:

$$\boldsymbol{x}_{J_i} = \sum_{\boldsymbol{x}' \in X_i} \boldsymbol{x}' \lambda_{\boldsymbol{x}'}, \text{ with } \boldsymbol{\lambda}(X_i) = 1, \text{ and } \boldsymbol{\lambda} \geq \boldsymbol{0}.$$

Now substituting for $\boldsymbol{x}_{J_i}$ in (3.17) yields an alternative linear programming formulation to (3.17) with a large but finite number of $\boldsymbol{\lambda}$-variables:

$$z^*_{(\boldsymbol{l},\boldsymbol{u})} = \max \quad \sum_{i=1}^{k} \sum_{\boldsymbol{x}\in X_i} (\boldsymbol{c}^T_{J_i}\boldsymbol{x})\lambda_{\boldsymbol{x}} \tag{3.19a}$$

$$\text{subject to} \quad \sum_{i=1}^{k} \sum_{\boldsymbol{x}\in X_i} (A^i\boldsymbol{x})\lambda_{\boldsymbol{x}} = \boldsymbol{b}_{I_0}, \tag{3.19b}$$

$$\boldsymbol{\lambda}(X_i) = 1, \qquad \forall i \in \{1,\dots,k\}, \tag{3.19c}$$

$$\boldsymbol{\lambda} \geq \boldsymbol{0}. \tag{3.19d}$$

The linear programming problem (3.19) is referred to as the *master problem* that we want to solve. The subscript $(\boldsymbol{l},\boldsymbol{u})$ is added to the optimal value z^* to stress the implicit dependence on the lower and upper bounds $\boldsymbol{l}$ and $\boldsymbol{u}$.

Due to the large number of $\boldsymbol{\lambda}$-variables it is difficult in general to solve (3.19) directly. Instead, we assume that we have at our disposal a relatively small subset $\bar{X}_i \subseteq X_i$ for each $i \in \{1,\dots,k\}$. In that case we can solve the following *restricted* master problem:

$$\max \quad \sum_{i=1}^{k} \sum_{\boldsymbol{x}\in \bar{X}_i} (\boldsymbol{c}^T_{J_i}\boldsymbol{x})\lambda_{\boldsymbol{x}} \tag{3.20a}$$

$$\text{subject to} \quad \sum_{i=1}^{k} \sum_{\boldsymbol{x}\in \bar{X}_i} (A^i\boldsymbol{x})\lambda_{\boldsymbol{x}} = \boldsymbol{b}_{I_0}, \tag{3.20b}$$

$$\boldsymbol{\lambda}(\bar{X}_i) = 1, \qquad \forall i \in \{1,\dots,k\}, \tag{3.20c}$$

$$\boldsymbol{\lambda}_{\bar{X}_i} \geq \boldsymbol{0}, \qquad \forall i \in \{1,\dots,k\}. \tag{3.20d}$$

Suppose that the restricted master problem is feasible and that we have an optimal primal-dual pair $(\boldsymbol{\lambda}, (\boldsymbol{\pi}, \boldsymbol{\mu}))$ to it, where $\boldsymbol{\pi} \in \mathbb{R}^{I_0}$ is associated with the constraints (3.20b), and $\boldsymbol{\mu} \in \mathbb{R}^k$ is associated with the constraints (3.20c). Obviously $\boldsymbol{\lambda}$ is a feasible solution to the master problem, and $(\boldsymbol{\lambda}, (\boldsymbol{\pi}, \boldsymbol{\mu}))$ is an optimal primal-dual pair to the master problem if it satisfies the reduced cost optimality conditions of Theorem 2.2. This is the case if for all $i \in \{1,\dots,k\}$, for all $\boldsymbol{x} \in X_i$, we have

$$\boldsymbol{c}^T_{J_i}\boldsymbol{x} - \boldsymbol{\pi}^T A^i\boldsymbol{x} - \mu_i \leq 0,$$

which is the case if

$$\zeta_i = \max\{(\boldsymbol{c}^{\boldsymbol{\pi}}_{J_i})^T\boldsymbol{x} \mid \boldsymbol{x} \in X_i\} \tag{3.21}$$

satisfies $\zeta_i \leq \mu_i$, where $\boldsymbol{c}^{\boldsymbol{\pi}} \in \mathbb{R}^n$ is defined by $\boldsymbol{c}^{\boldsymbol{\pi}}_{J_i} = \boldsymbol{c}_{J_i} - (\boldsymbol{\pi}^T A^i)^T$ for all $i \in \{1,\dots,k\}$. Problem (3.21) is called *pricing problem* i. If $\zeta_i > \mu_i$ for some $i \in \{1,\dots,k\}$, then the corresponding optimal solution $\boldsymbol{x}^*$ to pricing problem i is added to $\bar{X}_i$, and the master problem is resolved. A *column generation*

algorithm iteratively solves a restricted master problem, and pricing problems for all $i \in \{1, \ldots, k\}$, adding $\boldsymbol{\lambda}$-variables that violate the reduced cost optimality conditions of the master to the restricted master, until an optimal primal-dual pair to the master problem is found.

3.4.2 IP Column Generation

Now that we have seen how to apply Dantzig-Wolfe decomposition and column generation to linear programming problems, we will show how to adapt the LP-based branch-and-bound algorithm to integer linear programming problems that have a similar structure. For simplicity we concentrate on linear integer programming formulations that do not contain continuous variables. Consider the integer linear programming problem

$$\max\{z(\boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x} \mid a_{I_0}\boldsymbol{x} = \boldsymbol{b}_{I_0}, \boldsymbol{x} \in X\}, \tag{3.22}$$

where $\boldsymbol{b}, \boldsymbol{c}, A$ and I_0 are as in the previous section, and X is again defined in terms of $a_{I\setminus I_0}, \boldsymbol{b}_{I\setminus I_0}, \boldsymbol{l}$ and $\boldsymbol{u}$ by

$$P = \{\boldsymbol{x} \in \mathbb{R}^n \mid a_{I\setminus I_0}\boldsymbol{x} = \boldsymbol{b}_{I\setminus I_0}, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\}$$

and

$$X = P \cap \mathbb{Z}^n.$$

Note that (3.17) is an LP relaxation of (3.22).

Assuming that A is of the form (3.18) and after applying Dantzig-Wolfe decomposition to (3.22) we obtain the following equivalent integer linear programming formulation with a large but finite number of $\boldsymbol{\lambda}$-variables:

$$\begin{aligned}
\max \quad & \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in X_i} (\boldsymbol{c}_{J_i}^T \boldsymbol{x})\lambda_{\boldsymbol{x}} && && \text{(3.23a)}\\
\text{subject to} \quad & \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in X_i} (A^i\boldsymbol{x})\lambda_{\boldsymbol{x}} = \boldsymbol{b}_{I_0}, && && \text{(3.23b)}\\
& \boldsymbol{\lambda}(X_i) = 1, && \forall i \in \{1, \ldots, k\}, && \text{(3.23c)}\\
& \boldsymbol{\lambda}_{X_i} \in \{0,1\}^{X_i}, && \forall i \in \{1, \ldots, k\}, && \text{(3.23d)}
\end{aligned}$$

where $X_i \subseteq \mathbb{Z}^{J_i}$, and A^i are defined as before. Problem (3.23) is called the *IP master problem*, and in the context of IP column generation its LP relaxation (3.19) is called the *LP master problem*.

A *branch-and-price* algorithm for solving (3.22) is derived from the LP-based branch-and-bound algorithm as follows. The algorithm maintains a restricted LP master problem, that we assume to be feasible. The situation that the restricted LP master problem is infeasible is discussed in Section 3.4.4. As before, with each iteration j of the branch-and-price algorithm we associate a

unique set of bounds $\boldsymbol{l}^j, \boldsymbol{u}^j \in \mathbb{Z}^n$. In iteration j, the branch-and-price algorithm solves the LP master problem obtained from (3.17) by imposing the bounds $\boldsymbol{l}^j, \boldsymbol{u}^j$ instead of $\boldsymbol{l}, \boldsymbol{u}$. This is done by column generation. The formulation is obtained by imposing an upper bound of zero on all variables $\lambda_{\boldsymbol{x}}$ for which $\boldsymbol{l}^j_{J_i} \not\leq \boldsymbol{x}_{J_i}$ or $\boldsymbol{x}_{J_i} \not\leq \boldsymbol{u}^j_{J_i}$, where $\boldsymbol{x}_{J_i} \in \bar{X}_i, i \in \{1, \ldots, k\}$. The resulting pricing problems are of the form

$$\zeta_i = \max\{(\boldsymbol{c}^{\boldsymbol{\pi}}_{J_i})^T \boldsymbol{x} \mid \boldsymbol{x} \in X_i\}, \tag{3.24}$$

where for all $i \in \{1, \ldots, k\}$

$$X_i = P_i \cap \mathbb{Z}^{J_i},$$

and

$$P_i = \{\boldsymbol{x} \in \mathbb{R}^{J_i} \mid a_{I_i}\boldsymbol{x} = \boldsymbol{b}_{I_i}, \boldsymbol{l}^j_{J_i} \leq \boldsymbol{x} \leq \boldsymbol{u}^j_{J_i}\}.$$

If the pricing problems (3.24) are easy to solve, then we can use the column generation algorithm from Section 3.4.1 to solve the LP master problem in each iteration of the branch-and-bound algorithm. However, it may very well be that the pricing problems (3.24) are $\mathcal{NP}$-hard combinatorial optimisation problems. This is the case, for example, in Chapter 7 of this thesis, where we apply Dantzig-Wolfe decomposition to a vehicle routing problem, and the pricing problems are the problems of how to route and load the individual vehicles. Under such circumstances it would be rather optimistic to believe that we can solve the LP master problems to optimality by column generation even for medium size problems, as this would require us to solve numerous $\mathcal{NP}$-hard pricing problems in each iteration of the branch-and-price algorithm.

If we have an efficient heuristic for finding good feasible solutions to the pricing problems, we can use this heuristic to generate new columns instead of spending a lot of time in optimising the pricing problems. However, this approach gives us a new problem, namely, that a restricted LP master problem itself is not a relaxation of the IP master problem (3.23). The value of the restricted LP master problem is only an upper bound for the value of the IP master problem if the optimal LP solution to the restricted LP master problem is also optimal to the LP master problem. This is not necessarily the case when we generate columns by solving the pricing problems using primal heuristics as suggested above.

A solution to this problem was observed by Vanderbeck and Wolsey [115], and was successfully applied by Vanderbeck [114]. Focus on iteration $j \geq 1$ of the branch-and-price algorithm. Suppose we have solved a restricted master LP associated with iteration j that was feasible, and let $(\boldsymbol{\pi}, \boldsymbol{\mu}) \in \mathbb{R}^{I_1} \times \mathbb{R}^k$ be a dual solution corresponding to an optimal basis B. Vanderbeck and Wolsey observe that if we have

$$\bar{\zeta}_i \geq \max\{(\boldsymbol{c}^{\boldsymbol{\pi}}_{J_i})^T \boldsymbol{x} \mid \boldsymbol{x} \in X_i\}, \tag{3.25}$$

```
columnGenerationSolver(c', A', b', l, u, pricing)      // c', A' given implicitly,
{                                                      // returns λ*, z̄ ≥ z*_(l,u)
    let X be a subset of the column indices of A'; // e.g. X := ∅;
    do { solve LP z := max{(c'_X)^T λ_X | A'_X λ_X = b', λ_X ≥ 0}
        if LP infeasible {
            (X, feasible) := makeFeasible(A', b', l, u, X, pricing)
            if feasible {
                solve LP z := max{(c'_X)^T λ_X | A'_X λ_X = b', λ_X ≥ 0}; }
            else { return infeasible; }
        }
        let π be an optimal LP dual;
        (X', ζ, optimal) := pricing(π, l, u);
        z̄ := min(z̄, (b')^T (π + (0, ζ)^T));
        X := X ∪ X';
    } while X' ≠ ∅ and ζ ≠ 0;
    if optimal or ζ = 0 { z̄ := z; }
    let λ*_X be an optimal LP solution;
    return (λ*_X, z̄);
}
```

Algorithm 3.2: Solving LP by Column Generation in Branch-and-Price

where X_i is defined as before for all $i \in \{1, \ldots, k\}$, then the vector $(\boldsymbol{\pi}, \boldsymbol{\mu}+\bar{\boldsymbol{\zeta}})$ is a feasible solution to the dual linear program of the LP master problem associated with iteration j. From this, by Theorem 2.3 it follows that

$$z^*_{(\boldsymbol{l}^j, \boldsymbol{u}^j)} \leq \boldsymbol{b}^T \boldsymbol{\pi} + \mathbf{1}^T (\boldsymbol{\mu} + \bar{\boldsymbol{\zeta}}),$$

and that $\boldsymbol{\lambda}^*$ and $(\boldsymbol{\pi}, \boldsymbol{\mu})$ are optimal solutions to the primal and dual LP master problem, respectively, if $\bar{\boldsymbol{\zeta}} = \mathbf{0}$. Note that a value of $\bar{\boldsymbol{\zeta}}$ can be obtained by solving relaxations of the pricing problems, instead of the pricing problems themselves.

The IP master problem (3.23) can be restated as

$$\begin{aligned} \max \quad & (\boldsymbol{c}')^T \boldsymbol{\lambda} && \text{(3.26a)} \\ \text{subject to} \quad & A' \boldsymbol{\lambda} = \boldsymbol{b}, && \text{(3.26b)} \\ & \boldsymbol{\lambda} \geq \mathbf{0}, \text{ integer.} && \text{(3.26c)} \end{aligned}$$

Pseudo-code of the LP column generation algorithm that we employ in our branch-and-price code is given in Algorithm 3.2. Note that A' and $\boldsymbol{c}'$ are given implicitly, i.e., in such a way that we can compute $A'_{\boldsymbol{x}}$ and $c'_{\boldsymbol{x}}$ for each $\boldsymbol{x} \in \bigcup_{i=1}^k X_i$ once we have $\boldsymbol{x}$. The pricing problems depend on the original problem and are problem specific. Therefore, the column generation algorithm assumes as input a function pricing that it uses to solve the pricing problems. This function is of the form

$$\text{pricing} : \mathbb{R}^{m+k} \times \mathbb{R}^n \times \mathbb{R}^n \to 2^{\bigcup_{i=1}^k X_i} \times \mathbb{R}^k \times \{0, 1\},$$

and maps each triple $(\boldsymbol{\pi}, \boldsymbol{l}, \boldsymbol{u})$ to a triple $(X', \boldsymbol{\zeta}, f)$, where $\boldsymbol{\pi}$ is an optimal dual to the restricted LP master, $\boldsymbol{l}, \boldsymbol{u}$ are bounds, X' is a set of column indices that correspond to $\boldsymbol{\lambda}$ variables that violated the reduced cost optimality criteria of the LP master problem, $\boldsymbol{\zeta}$ is an upper bound on the value of the pricing problems, and $f = 1$ if the pricing problems were solved to optimality, and $f = 0$ otherwise. Furthermore, the algorithm makes use of a function makeFeasible. Given a subset of the column indices X', bound vectors $\boldsymbol{l}, \boldsymbol{u}$, and the pricing function, makeFeasible determines whether there exists a set of column indices $X'' \supseteq X'$ such that $A'_{X''} = \boldsymbol{b}'$ is feasible. The function makeFeasible returns a pair (X'', f), where f is 1 if $A'_{X''} = \boldsymbol{b}'$ and 0 otherwise.

3.4.3 The Branching Scheme

After we have solved the LP relaxation in some iteration i of the branch-and-price algorithm, we have at our disposal a vector $\boldsymbol{\lambda}^*$ that is a feasible solution to the LP master problem, and what we want to have is a feasible solution to our original problem (3.22). It is undesirable to branch on the $\boldsymbol{\lambda}$-variables, as this changes the structure of the pricing problems in all nodes that are not the root node of the branch-and-bound tree. Indeed, we would have to prevent $\boldsymbol{x}$ for which $\lambda_{\boldsymbol{x}}$ has been set to either zero or one to be regenerated by a pricing problem. For the same reason we do not set $\boldsymbol{\lambda}$-variables based on reduced cost criteria.

Several alternative branching schemes are reported in the literature. Here, we discuss a scheme that is known as branching *on original variables* (see e.g. Barnart *et al.* [14]). In this scheme, we compute $\boldsymbol{x}^* \in \mathbb{R}^n$ given by

$$\boldsymbol{x}^* = \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in X_i} \boldsymbol{x} \lambda^*_{\boldsymbol{x}},$$

which is the same solution as $\boldsymbol{\lambda}^*$ but restated in terms of the variables of the original integer programming problem (3.22). If $\boldsymbol{x}^* \in \mathbb{Z}^n$, we are done. Otherwise we can select an index $j^* \in \{1, \ldots, n\}$ such that $x^*_{j^*}$ is fractional and partition the feasible region by enforcing lower and upper bounds $(\boldsymbol{l}^i, \tilde{\boldsymbol{u}}^i)$ on one branch and $(\tilde{\boldsymbol{l}}^i, \boldsymbol{u}^i)$ on the other branch, where $\tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i \in \mathbb{Z}^n$ are given by

$$\tilde{l}^i_j = \begin{cases} \lceil x^*_j \rceil, & \text{if } j = j^*, \\ l^i_j & \text{otherwise,} \end{cases} \qquad \text{and} \qquad \tilde{u}^i_j = \begin{cases} \lfloor x^*_j \rfloor, & \text{if } j = j^*, \\ u^i_j & \text{otherwise.} \end{cases}$$

In our implementation the index j^* is selected using the Padberg-Rinaldi rule (see Section 3.2.4).

We end this section by remarking that for the correctness of the branch-and-price algorithm it is necessary to branch if $\boldsymbol{x}^*$ is integer but one does not have a proof that $\boldsymbol{x}^*$ is optimal. This can be done in a similar way as above.

```
makeFeasible(A', b', l, u, X, pricing)        // returns (X̃, f): f = 1 iff
{                                             // A'_X̃ λ_X̃ = b', λ_X̃ ≥ 0 is feasible
    c := 0;                                   // implicitly used by pricing
    do { solve z := max{−s | A'_X λ_X + b' s = b', λ ≥ 0, s ≥ 0};
         if z = 0 { return (X, 1); }
         let π be an optimal dual solution;
         (X', ζ) := pricing(π, l, u);         // solve to optimality
         X := X ∪ X';
    } while X ≠ ∅ and ζ ≠ 0;
    return (X, 0);
}
```

Algorithm 3.3: Searching a Feasible Set of Columns

3.4.4 Infeasible Nodes

Consider iteration $i > 1$ of the branch-and-price algorithm, and observe that there is no reason to assume that the restricted LP master problem is feasible after we impose the bounds $(\boldsymbol{l}^i, \boldsymbol{u}^i)$. Moreover, the infeasibility of the restricted LP master problem does not imply the infeasibility of the LP master problem associated with iteration i as it contains only a subset of the $\boldsymbol{\lambda}$-variables. It follows that our branch-and-price algorithm is not correct unless we have a way of either proving that the LP master problem associated with iteration i is infeasible, or finding a feasible solution to it.

A possible solution here is to use the so-called "big M" method that is used for finding an initial feasible basis in linear programming. In this method one adds slack variables that have a symbolic cost coefficient that is interpreted as a large negative value (and usually denoted by M). Unfortunately, this would require us to write subroutines for solving the pricing problems that can handle symbolic values, which is undesirable. A solution suggested by Barnhart *et al.* [14] is to use slack variables with real-valued negative cost coefficients instead. For such a scheme to be correct one needs a lower bound on the cost of the slack variables that suffices to prove infeasibility. Moreover, this lower bound should be of such a small magnitude in absolute value that the pricing algorithms can manipulate the resulting dual solutions without numerical problems.

If the objective function is zero, then any negative cost coefficient suffices to prove infeasibility. This way, one obtains a well defined two-phase approach for LP column generation that is similar to the two-phase approach that is used for linear programming (see e.g. Chvátal [28, Chapter 8]). Suppose we have an infeasible restricted LP master problem of the form

$$\begin{aligned} \max \quad & (\boldsymbol{c}'_X)^T \boldsymbol{\lambda}_X && \text{(3.27a)} \\ \text{subject to} \quad & A'_X \boldsymbol{\lambda}_X = \boldsymbol{b}', && \text{(3.27b)} \\ & \boldsymbol{\lambda}_X \geq \boldsymbol{0}. && \text{(3.27c)} \end{aligned}$$

To find a feasible solution to (3.27), we construct an auxiliary master linear program by replacing the cost vector $\boldsymbol{c}$ by the zero vector and adding a single artificial variable $s \in \mathbb{R}$ with cost -1, and column $\boldsymbol{b}'$:

$$\begin{aligned} \max \quad & z(\boldsymbol{\lambda}, s) = -s && \text{(3.28a)} \\ \text{subject to} \quad & A'\boldsymbol{\lambda} + \boldsymbol{b}'s = \boldsymbol{b}', && \text{(3.28b)} \\ & \boldsymbol{\lambda} \geq \boldsymbol{0}, s \geq 0. && \text{(3.28c)} \end{aligned}$$

Note that $(\boldsymbol{\lambda}, s) := (\boldsymbol{0}, 1)$ is a feasible solution to (3.28) with value -1. Now, if $\boldsymbol{\lambda}^*$ is a feasible solution to (3.27) then $(\boldsymbol{\lambda}^*, 0)$ is a feasible solution to (3.28) with value 0. Conversely, if $(\boldsymbol{\lambda}^*, s^*)$ is a feasible solution to (3.28) with value 0 then $s^* = 0$, so $\boldsymbol{\lambda}^*$ is a feasible solution to (3.27). Pseudo-code for a procedure that solves (3.28) by column generation to either prove infeasibility of the LP master problem, or to find a set of column indices X that defines a feasible LP master problem can be found in Algorithm 3.3. Note that in order to prove that the LP master problem associated with iteration i is infeasible, it is necessary to solve the pricing problems to optimality.

3.5 Branch-Price-and-Cut

Consider again the integer linear programming problem

$$\max\{z(\boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x} \mid a_{I_0}\boldsymbol{x} = \boldsymbol{b}_{I_0}, \boldsymbol{x} \in X\}, \tag{3.29}$$

where $\boldsymbol{b}, \boldsymbol{c}, A$ and I_0 are as in the previous section, and X is again defined in terms of $a_{I\setminus I_0}, \boldsymbol{b}_{I\setminus I_0}, \boldsymbol{l}$ and $\boldsymbol{u}$ by

$$P = \{\boldsymbol{x} \in \mathbb{R}^n \mid a_{I\setminus I_0}\boldsymbol{x} = \boldsymbol{b}_{I\setminus I_0}, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\}$$

and

$$X = P \cap \mathbb{Z}^n.$$

Assume that A is of the form (3.18). We again apply Dantzig-Wolfe decomposition, but this time we leave $\boldsymbol{x}_{J_1}$ aside and only reformulate $\boldsymbol{x}_{J_i}$ for $i \in \{2, \ldots, k\}$. In this way we obtain the following equivalent integer linear programming problem with a large but finite number of $\boldsymbol{\lambda}$-variables:

$$\begin{aligned} \max \quad & \boldsymbol{c}_{J_1}^T\boldsymbol{x}_{J_1} + \sum_{i=2}^{k}\sum_{\boldsymbol{x}\in X_i}(\boldsymbol{c}_{J_i}^T\boldsymbol{x})\lambda_{\boldsymbol{x}} && && \text{(3.30a)} \\ \text{subject to} \quad & A^1\boldsymbol{x}_{J_1} + \sum_{i=2}^{k}\sum_{\boldsymbol{x}\in X_i}(A^i\boldsymbol{x})\lambda_{\boldsymbol{x}} = \boldsymbol{b}_{I_0}, && && \text{(3.30b)} \\ & \boldsymbol{l}_{J_1} \leq \boldsymbol{x}_{J_1} \leq \boldsymbol{u}_{J_1},\ \boldsymbol{x}_{J_1} \text{ integer}, && && \text{(3.30c)} \\ & \boldsymbol{\lambda}(X_i) = 1, && \forall i \in \{2, \ldots, k\}, && \text{(3.30d)} \\ & \boldsymbol{\lambda}_{X_i} \in \{0, 1\}^{X_i}, && \forall i \in \{2, \ldots, k\}, && \text{(3.30e)} \end{aligned}$$

where $X_i \subseteq \mathbb{Z}^{J_i}$, and A_i are defined as before. The resulting model can be solved using the branch-and-price algorithm of the previous section, allowing for branching on the $\boldsymbol{x}_{J_1}$-variables as described in Section 3.2.4, and branching on the original $\boldsymbol{x}_{J_i}$-variables for $i \in \{2, \ldots, k\}$ as described in Section 3.4.3.

If we have one or more classes of valid inequalities for $\text{conv}(X_1)$ at our disposal, then we can add these to strengthen the LP formulation in each iteration of the branch-and-price algorithm. Furthermore, one may apply variable setting based on reduced cost criteria to the $\boldsymbol{x}_{J_1}$-variables whenever we have a proof that the bound obtained from the restricted LP master problem is valid for the LP master problem. We will call the resulting algorithm a *branch-price-and-cut* algorithm. A branch-price-and-cut algorithm applies a combined cutting plane and column generation algorithm to solve the LP relaxations in each iteration. In our implementation we alternate between rounds of column generation and separation until the termination criteria of both the column generation algorithm and the cutting plane algorithm have been satisfied in two consecutive rounds.

Chapter 4

Maximum Independent Set

4.1 Introduction

Given an undirected graph $G = (V, E)$, an *independent set* in G is a subset $S \subseteq V$ such that no two nodes in S are connected in G, i.e. for all edges $\{u, v\} \in E$ we have that $\{u, v\} \not\subseteq S$. The *maximum independent set problem* is the problem of finding an independent set of maximum cardinality in a given graph. Independent sets are sometimes called *stable sets*, *vertex packings*, or *node packings*.

The *complement* of G has the same node set, and is denoted by $\bar{G} = (V, \bar{E})$ where $\{u, v\} \in \bar{E}$ if and only if $\{u, v\} \notin E$. A *clique* in G is a set of nodes $S \subseteq V$ such that for all $\{u, v\} \in S$, $\{u, v\} \in E$. A subset $S \subseteq V$ is an independent set in G if and only if S is a clique in $\bar{G}$. The *maximum clique problem* is the problem of finding a clique of maximum cardinality in a given graph. Due to the equivalence between independent sets in a graph and cliques in its complement, any algorithm for finding a maximum independent set also gives an algorithm for finding a maximum clique, and vice versa.

Also, independent set problems are related to so-called *set packing problems*, in which one is given a ground set and a collection of subsets of this ground set, and one has to find a sub-collection of the given subsets that are disjoint and maximal with respect to some objective function. The reduction from the set packing problem to the independent set problem uses a conflict graph that is similar in spirit as the one that is used in Section 3.2.3. For more details on set packing and its relation to independent set problems we refer to Borndörfer [20] and Marchand, Martin, Weismantel, and Wolsey [82].

4.1.1 Motivation, Literature, and Chapter Outline

Several optimisation algorithms for finding maximum independent sets [101, 106, 81, 80, 85, 126, 48] and maximum cliques [12, 6, 7, 26, 13] have been reported on in the literature. Moreover, several real-life problems can be formulated as independent set problems [100, 2, 117, 129]. The polyhedral structure of the

independent set polytope has been reported on by various authors [90, 86, 87, 112].

Our interest in independent set problems originates from the fact that map labelling problems, which are the subject of Chapter 5, can be reformulated as independent set problems. We develop a number of powerful techniques for the problem that are meant to work well on sparse large-size instances. We do not claim that our independent set code is the best code available for solving independent set problems. It will, however, provide us with the best optimisation algorithm for map labelling problems that is currently known.

In this chapter we explain all the ingredients of our branch-and-cut algorithm for maximum independent set. Other LP-based branch-and-bound algorithms for maximum independent set reported on in the literature are the cutting plane/branch-and-bound algorithm by Nemhauser and Sigismondi [85] and the branch-and-cut algorithm of Rossi and Smriglio [101]. Our approach differs from Nemhauser and Sigismondi in that we use a branch-and-cut approach with a different scheme for lifting odd hole inequalities, different primal heuristics, and stronger implications and pre-processing techniques.

We will first introduce several integer programming formulations for the maximum independent set problem in Section 4.1.2, and recall some basic notions in Section 4.1.3. Using these concepts, we will discuss LP rounding algorithms for the problem in Section 4.2, and then present our branch-and-cut algorithm in Section 4.3. In Section 4.4 we evaluate the performance of our independent set code on a class of randomly generated test instances that is commonly used for generating test instances in the literature.

4.1.2 Integer Programming Formulations

To obtain an integer programming formulation for the independent set problem, we use the decision variables $\boldsymbol{x} \in \{0,1\}^V$, where for node $v \in V$ $x_v = 1$ if v is in the independent set, and $x_v = 0$ if v is not. Given an undirected graph $G = (V, E)$, let

$$P_E = \{\boldsymbol{x} \in \mathbb{R}^V \mid \boldsymbol{x}(\{u,v\}) \leq 1 \text{ for all } \{u,v\} \in E, \boldsymbol{x} \geq \boldsymbol{0}\}.$$

The convex hull of incidence vectors of all independent sets in G is denoted by

$$P_{\text{IS}} = \text{conv}(P_E \cap \{0,1\}^V).$$

The maximum independent set problem can be formulated as the following integer programming problem:

$$\max\{\boldsymbol{x}(V) \mid \boldsymbol{x} \in P_E \cap \{0,1\}^V\}. \tag{4.1}$$

We call this formulation the *edge formulation* of the maximum independent set problem.

Let $C \subseteq V$ be a clique in G. It follows directly from the definition of independent sets that any independent set can have at most one node from C.

Hence, the *clique inequality*

$$\boldsymbol{x}(C) \leq 1 \tag{4.2}$$

is a valid inequality for P_{IS} for every clique C. If C is maximal, we call (4.2) a *maximal* clique inequality. Padberg [90] showed that the clique inequality on C is facet-defining for P_{IS} if and only if C is a maximal clique. Let $\mathcal{C}$ be a collection of not necessarily maximal cliques in G such that for each edge $\{u, v\} \in E$, there exists a clique $C \in \mathcal{C}$ with $\{u, v\} \subseteq C$, and let

$$P_{\mathcal{C}} = \{\boldsymbol{x} \in \mathbb{R}^V \mid \boldsymbol{x}(C) \leq 1 \text{ for all } C \in \mathcal{C}, \boldsymbol{x} \geq \boldsymbol{0}\}.$$

Then, problem (4.1) can be reformulated as

$$\max\{\boldsymbol{x}(V) \mid \boldsymbol{x} \in P_{\mathcal{C}} \cap \{0,1\}^V\}. \tag{4.3}$$

We call this formulation the *clique formulation* of the maximum independent set problem.

Finally, we can rewrite the clique formulation by introducing the slack variables $\boldsymbol{s} \in \{0,1\}^{\mathcal{C}}$. This has as advantage that we can use the clique inequalities for GUB branching. Let

$$P_{\mathcal{C}}^{\text{GUB}} = \{(\boldsymbol{x}, \boldsymbol{s}) \in \mathbb{R}^{V \cup \mathcal{C}} \mid \boldsymbol{x}(C) + s_C = 1 \text{ for all } C \in \mathcal{C}, \boldsymbol{x} \geq \boldsymbol{0}, \boldsymbol{s} \geq \boldsymbol{0}\}.$$

This leads to the following formulation of the maximum independent set problem, that we call the *GUB* formulation:

$$\max\{\boldsymbol{x}(V) \mid (\boldsymbol{x}, \boldsymbol{s}) \in P_{\mathcal{C}}^{\text{GUB}} \cap \{0,1\}^{V \cup \mathcal{C}}\}. \tag{4.4}$$

In our branch-and-cut algorithm we use the GUB formulation.

The sets $P_E, P_{\mathcal{C}}$, and $P_{\mathcal{C}}^{\text{GUB}}$ are related as follows:

Proposition 4.1. *Let $\mathcal{C}$ be a collection of cliques such that there exists $C \in \mathcal{C}$ with $\{u, v\} \subseteq C$ for each $\{u, v\} \in E$. Then, $P_{IS} \subseteq \{\boldsymbol{x}_V \mid \boldsymbol{x} \in P_{\mathcal{C}}^{GUB}\} = P_{\mathcal{C}} \subseteq P_E$.*

Some of the sub-routines we use in our branch-and-cut algorithm assume as input a vector $\boldsymbol{x} \in P_E$. By Proposition 4.1 we can apply them to $\boldsymbol{x}_V$, where $\boldsymbol{x} \in P_{\mathcal{C}}^{\text{GUB}}$ is obtained by maximising over the GUB formulation.

4.1.3 Some Basic Notions from Graph Theory

To present the algorithms in this chapter, we will use some basic notions from graph theory. Let $G = (V, E)$ be a graph. For $U \subseteq V$, let $N(U)$ denote the set of *neighbours* of U in G, i.e.,

$$N(U) = \{v \in V \setminus U \mid \{u, v\} \in \delta(U)\}.$$

For singleton sets $\{u\}$ we will abbreviate $N(\{u\})$ to $N(u)$. For any natural number k, the *k-neighbourhood* of a set of nodes $S \subseteq V$ in a graph G, denoted

by $N_k(S)$, consists of all nodes in G that can be reached from a node in S by traversing at most k edges, i.e.,

$$N_k(S) = \begin{cases} N(T) \cup T, & \text{where } T = N_{k-1}(S), \quad \text{if } k > 0, \text{ and} \\ S, & \text{if } k = 0. \end{cases}$$

For singleton sets $\{v\}$ we will abbreviate $N_k(\{v\})$ to $N_k(v)$.

Let $G = (V, E)$ be a graph. The *length* of a path P in G is the number of edges $|E(P)|$ in P. The *diameter* of G, denoted by $\mathrm{diam}(G)$, is the maximum length of a shortest path connecting two nodes in G, i.e.,

$$\mathrm{diam}(G) = \max_{u,v \in V} \min\{|E(P)| \mid \text{P is a path from } u \text{ to } v \text{ in } G\}.$$

For $S \subseteq V$, the *graph induced by* S, denoted by $G[S]$, is the graph with node set S and edge set $E(S)$. A *connected component* in G is a connected induced graph $G[S]$ that is maximal with respect to inclusion of nodes, where $S \subseteq V$.

4.2 Heuristics for Maximum Independent Set

Here we consider computationally cheap heuristics for finding hopefully large independent sets, starting from an LP solution in P_E. These heuristics are reported on by Strijk, Verweij and Aardal [107]. In Section 4.2.1 and 4.2.2 we discuss LP rounding heuristics, and in Section 4.2.3 we discuss simple local search neighbourhoods that can be used to increase the cardinality of the independent sets produced by the rounding heuristics.

4.2.1 Simple LP Rounding

Suppose we are given a point $\boldsymbol{x} \in P_E$. Let S be the set of nodes that correspond to $\boldsymbol{x}$-variables with a value strictly greater than one half, i.e., $S = \{v \in V \mid x_v > \frac{1}{2}\}$. Because $\boldsymbol{x} \in P_E$, we know that for each edge $\{u, v\} \in E$ at most one of u and v can have a value strictly greater than one half and thus be in S. It follows that S is an independent set in G. The simple LP rounding algorithm rounds $\boldsymbol{x}_S$ up to $\mathbf{1}$, and $\boldsymbol{x}_{V \setminus S}$ down to $\mathbf{0}$ to obtain the vector $\boldsymbol{x}' = \chi^S \in P_E \cap \{0,1\}^V$, which it returns as a solution. This can be done in $O(|V|)$ time.

The quality of the solution of the LP rounding algorithm can be arbitrarily bad. For instance, the vector $\boldsymbol{x} \in P_E$ with $x_v = \frac{1}{2}$ for all $v \in V$ is rounded to $\mathbf{0}$, corresponding to the empty independent set. On the other hand, the algorithm is capable of producing any optimal solution. For example, if $\boldsymbol{x} = (1-\epsilon)\chi^{S^*} + \epsilon\chi^{V \setminus S^*}$ for some maximum independent set S^* and any ϵ such that $0 \leq \epsilon < \frac{1}{2}$, then the algorithm produces χ^{S^*} as an answer, an optimal solution.

Nemhauser and Trotter [86] have shown that if $\boldsymbol{x}$ is obtained by maximising any objective function over P_E, then $\boldsymbol{x}$ is a vector with components 0, $\frac{1}{2}$, and 1 only. In this case the simple LP rounding algorithm degenerates to selecting the components of $\boldsymbol{x}$ with value 1.

4.2.2 Minimum Regret Rounding

Denote by $\mathcal{I}$ the collection of all independent sets in G. Suppose we are given a vector $\boldsymbol{x} \in P_E$ with some fractional components. We are going to round $\boldsymbol{x}$ by repeatedly choosing an independent set $I \in \mathcal{I}$ with $\mathbf{0} < \boldsymbol{x}_I < \mathbf{1}$, rounding $\boldsymbol{x}_I$ up to $\mathbf{1}$, and $\boldsymbol{x}_{N(I)}$ down to $\mathbf{0}$, until $\boldsymbol{x}$ is integral. Let $I \in \mathcal{I}$. The rounding operation above defines a function f that maps $\boldsymbol{x}$ on a vector $\boldsymbol{x}' \in P_E$ using I:

$$f : P_E \times \mathcal{I} \to P_E : (\boldsymbol{x}, I) \mapsto \boldsymbol{x}', \qquad \text{where } x'_u = \begin{cases} 1 & \text{if } u \in I, \\ 0 & \text{if } u \in N_1(I), \\ x_u & \text{otherwise.} \end{cases}$$

We say that f *rounds up* $\boldsymbol{x}_I$.

Lemma 4.2. *Let $\boldsymbol{x} \in P_E$ and $I \in \mathcal{I}$. Then $f(\boldsymbol{x}, I) \in P_E$.*

Proof. Let $\boldsymbol{x}' = f(\boldsymbol{x}, I)$. Since I is an independent set and by construction of f, $\boldsymbol{x}'_{N_1(v)\setminus\{v\}} = \mathbf{0}$ for all $v \in I$, we have that $\boldsymbol{x}'_{N_1(I)} \in P_E$. Since $\mathbf{0} \leq \boldsymbol{x}'_{V\setminus I} \leq \boldsymbol{x}_{V\setminus I}$, we have $\boldsymbol{x}'_{V\setminus I} \in P_E$. The proof follows because by definition of $N_1(I)$ there do not exist edges $\{u, v\}$ with $u \in V \setminus N_1(I)$ and $v \in I$. □

We first study the effect of f on the objective function $\boldsymbol{x}(V)$. Again, let $I \in \mathcal{I}$ and $\boldsymbol{x}' = f(\boldsymbol{x}, I)$. Define the function $r : P_E \times \mathcal{I} \to \mathbb{R}$ as the difference in objective function value between $\boldsymbol{x}$ and $\boldsymbol{x}'$:

$$\begin{aligned} r(\boldsymbol{x}, I) &= \sum_{v \in V} x_v - \sum_{v \in V} (f(\boldsymbol{x}, I))_v = \sum_{v \in V} (x_v - x'_v) = \sum_{v \in N_1(I)} x_v - |I| \\ &= \boldsymbol{x}(N_1(I)) - |I|. \end{aligned}$$

Because we will later apply f to a vector $\boldsymbol{x} \in P_E$ that is optimal with respect to $\boldsymbol{x}(V)$, we have that $\boldsymbol{x}(V) \geq \boldsymbol{x}'(V)$, so $r(\boldsymbol{x}, I) \geq 0$ is the degradation of the objective function in that case. Since we do not like degradation, we call $r(\boldsymbol{x}, I)$ the *regret* we have when rounding $\boldsymbol{x}_I$ to $\mathbf{1}$.

Now take $\boldsymbol{x} \in P_E, I \in \mathcal{I}$ with $|I| > 1$, and choose non-empty sets $I_1, I_2 \subset I$ where $I_1 = I \setminus I_2$. Then,

$$\begin{aligned} r(\boldsymbol{x}, I) &= \boldsymbol{x}(N_1(I)) - |I| \\ &= \boldsymbol{x}(N_1(I_1)) + \boldsymbol{x}(N_1(I_2)) - \boldsymbol{x}(N_1(I_1) \cap N_1(I_2)) - |I_1| - |I_2| \\ &= r(\boldsymbol{x}, I_1) + r(\boldsymbol{x}, I_2) - \boldsymbol{x}(N_1(I_1) \cap N_1(I_2)). \end{aligned}$$

This shows that if

$$\boldsymbol{x}(N_1(I_1) \cap N_1(I_2)) = 0, \tag{4.5}$$

then the regret of rounding $\boldsymbol{x}_I$ to $\mathbf{1}$ is the same as the combined regret of rounding $\boldsymbol{x}_{I_1}$ and $\boldsymbol{x}_{I_2}$ to $\mathbf{1}$. It follows that we can restrict our choice of I to independent sets that cannot be partitioned into subsets I_1, I_2 satisfying

condition (4.5). This is the case if and only if the graph induced by the support of $\boldsymbol{x}_{N_1(I)}$ is connected.

If we choose I in such a way that $\mathbf{0} < \boldsymbol{x}_I < \mathbf{1}$, then $f(\boldsymbol{x}, I)$ has at least $|I|$ fewer fractional components than $\boldsymbol{x}$. We will use this to define a greedy rounding algorithm as follows. The algorithm has as input a vector $\boldsymbol{x} \in P_E$ and an integer $t > 0$ and repeatedly replaces $\boldsymbol{x}$ by $f(\boldsymbol{x}, I)$ for some set I, rounding $\boldsymbol{x}_I$ to $\mathbf{1}$. This is done in t phases, numbered $t, t-1, \ldots, 1$. In phase k, we only work with sets $I \in \mathcal{I}$ satisfying

$$|I| = k, \tag{4.6}$$

$$\mathbf{0} < \boldsymbol{x}_I < \mathbf{1}, \text{ and} \tag{4.7}$$

$$G[\operatorname{supp}(\boldsymbol{x}_{N_1(I)})] \text{ is connected.} \tag{4.8}$$

During phase k, the next set $I \in \mathcal{I}$ is chosen so as to minimise the regret $r(\boldsymbol{x}, I)$ within these restrictions. Phase k terminates when there are no more sets $I \in \mathcal{I}$ satisfying these conditions.

We name this algorithm the *minimum regret rounding algorithm* after the choice of I. Note that at any time during the algorithm, $\boldsymbol{x}_F$ is an optimal LP solution to the maximum independent set problem in $G[F]$ if the original vector $\boldsymbol{x}$ was one in G, where $F = \{v \in V \mid 0 < x_v < 1\}$ is the fractional support of $\boldsymbol{x}$. It follows that the value $\boldsymbol{x}(V)$ never increases over any execution of the algorithm.

Phase k is implemented in iterations as follows. We maintain a priority queue Q that initially contains all sets $I \in \mathcal{I}$ satisfying conditions (4.6)–(4.8), where the priority of set I is the value of $r(\boldsymbol{x}, I)$. In each iteration, we extract a set I from Q with minimum regret. If $\boldsymbol{x}_I$ has integral components or if the graph induced by the support of $\boldsymbol{x}_{N_1(I)}$ is not connected, then we proceed with the next iteration. Otherwise we update Q. This is done by decreasing the priorities of all $I' \in Q$ with $N_1(I') \cap N_1(I) \neq \varnothing$ by $\boldsymbol{x}(N_1(I') \cap N_1(I))$. We replace our current vector $\boldsymbol{x}$ by $f(\boldsymbol{x}, I)$ and proceed with the next iteration. Phase k terminates when Q is empty.

Lemma 4.3. *Let $F \subseteq V$ be the fractional support of $\boldsymbol{x}$ upon termination of phase k of the minimum regret rounding algorithm. For any $F' \subseteq F$ such that $G[F']$ is a connected component of $G[F]$,*

$$\operatorname{diam}(G[F']) < 2(k-1).$$

Proof. Let $G[F']$ be a connected component of $G[F]$ and suppose, by way of contradiction, that the graph $G[F']$ has a diameter of at least $2(k-1)$. Then, there exists nodes $u, v \in F'$ for which the shortest path P in $G[F']$ has length exactly $2(k-1)$. Let

$$P = (u = v_0, e_1, v_1, \ldots, e_{2(k-1)}, v_{2(k-1)} = v).$$

Consider the set $I = \{v_0, v_2, \ldots, v_{2(k-1)}\}$. Observe that $I \subset F'$. We argue that I is an independent set. Suppose for some $v_i, v_j \in I$ we had $\{v_i, v_j\} \in E$, thus

$\{v_i, v_j\} \in E(F')$. Since we can assume without loss of generality that $i < j$, P can be shortened by replacing the sequence $v_i, e_{i+1}, v_{i+1}, \ldots, e_j, v_j$ by the sequence $v_i, \{v_i, v_j\}, v_j$, and still be a path in $G[F']$. As this contradicts our choice of P it follows that no such $v_i, v_j \in I$ exist. Thus I is an independent set in $G[F']$. Because $|I| = k$, $F' \subseteq F$, and $G[F']$ is connected, I satisfies conditions (4.6)–(4.8). This contradicts the termination of phase k. □

Theorem 4.4. *Let $\boldsymbol{x} = \chi^S$ be the vector returned by the minimum regret rounding algorithm with input $\boldsymbol{x}'$ and some $t > 0$, where $\boldsymbol{x}' \in P_E$. Then S is an independent set.*

Proof. Since $\boldsymbol{x}$ is obtained from $\boldsymbol{x}'$ by iteratively applying f, we have $\boldsymbol{x} \in P_E$ by Lemma 4.2. From Lemma 4.3, we have that upon termination of phase $k = 1$, the diameter of each connected components of the fractional support of $\boldsymbol{x}$ is strictly less than 0. This implies that the graph induced by the fractional support is empty, hence $\boldsymbol{x}$ is an integer vector. □

We have implemented the minimum regret rounding heuristic for $t = 1$ and $t = 2$. Let us analyse the time complexity of the minimum regret rounding algorithm for those values of t. We start by analysing the time complexity of phase $k = 1$, the only phase of the algorithm if $t = 1$. In this phase, condition (4.8) is automatically fulfilled, and any fractional component x_v defines its own singleton independent set in Q. Since the regret of the set $\{v\}$ can be computed in $O(|\delta(v)| + 1)$ time for each $v \in V$, Q can be initialised in $O(|V| \log |V| + |E|)$ time. Extracting a node with minimum regret from Q takes at most $O(\log |V|)$ time. Moreover, for each node $v \in V$, x_v is set to 0 at most once, and when this happens at most $|\delta(v)|$ priorities have to be decreased. Since decreasing a priority takes $O(\log |V|)$ time, the total time spent in decreasing priorities is at most

$$\sum_{v \in V} (|\delta(v)| \cdot O(\log |V|)) = O(\log |V|) \cdot \sum_{v \in V} |\delta(v)| = O(|E| \log |V|).$$

Summing everything together, phase $k = 1$ of the rounding algorithm can be implemented to work in $O((|V| + |E|) \log |V|)$ time.

Next we analyse the time complexity of phase $k = 2$, which precedes phase $k = 1$ in the case that $t = 2$. Each node $v \in V$ occurs in at most $|V| - 1$ independent sets of size two, and $N_1(v)$ intersects with $N_1(I)$ for at most $|\delta(v)| \cdot (|V| - 1)$ possible choices of $I \in \mathcal{I}$ with $|I| = 2$. So, the number of independent sets of size two is at most $O(|V|^2)$, and their regret values can be initialised in time

$$O(\sum_{v \in V} |\delta(v)| \cdot (|V| - 1)) = O(|V||E|).$$

It follows that Q can be initialised in $O(|V|^2 \log |V| + |V||E|)$ time. For $v \in V$, when x_v is set to 0, at most $|\delta(v)| \cdot (|V| - 1)$ priorities have to be decreased, each decrease of a priority taking $O(\log |V|)$ time, summing up to a total of

$O(|V||E|\log|V|)$ time. Finally, we have to check condition (4.8). This can be done by keeping track of the sizes of the sets $\mathrm{supp}(\boldsymbol{x}_{N_1(v)\cap N_1(w)})$ for each $\{v,w\}\in\mathcal{I}$ $(v\neq w)$. These sizes can be stored in a $|V|\times|V|$ matrix. This matrix has to be updated each time we set x_v to 0 for some $v\in V$. This takes at most $O(|V||E|)$ time in total. It follows that phase $k=2$ of the rounding algorithm can be implemented in $O((|V|+|E|)|V|\log|V|)$ time. This term dominates the time complexity of the algorithm.

We complete this section with observing that for any natural number $k>2$, the number of different independent sets of size k is at most $|V|^k$, which is polynomial in $|V|$ for fixed k. As a consequence, implementing the minimum regret rounding algorithm in a brute-force fashion will yield a polynomial algorithm for any fixed value of t.

4.2.3 Iterative Improvement

Here we consider local search neighbourhoods and iterative improvement for the maximum independent set problem.

Definition 4.1. Let S be an independent set in G, and $k\geq 1$ be an integer. The *k-opt* neighbourhood of S, denoted by $\mathcal{N}_k(S)$, is defined as the collection of independent sets S' of cardinality $|S'|=|S|+1$ that can be obtained from S by removing $k-1$ nodes and adding k nodes.

So, if $S'\in\mathcal{N}_k(S)$, then $S'=(S\setminus U)\cup W$, for some U and W with $U\subseteq S$, $|U|=k-1$, $W\subset(V\setminus S)\cup U$, and $|W|=k$. The k-opt neighbourhood is undefined if $|S|<k-1$. Because we do not require that $U\cap W=\varnothing$ we have that, if $\mathcal{N}_k(S)$ is defined, then $\mathcal{N}_j(S)\subseteq\mathcal{N}_k(S)$ for all $j\in\{1,\ldots,k\}$.

Definition 4.2. An independent set S is *k-optimal* if the k-opt neighbourhood of S is empty.

There is no guarantee that the minimum regret rounding algorithms produce k-optimal independent sets for any $k\geq 1$. This motivates our interest in the k-opt neighbourhoods of independent sets.

Proposition 4.5. *If an independent set S is k-optimal, then S is l-optimal for all $l\in\{1,\ldots,k\}$.*

Proof. The proposition holds because $\mathcal{N}_l(S)\subseteq\mathcal{N}_k(S)$ for all $l\in\{1,\ldots,k\}$. □

The k-opt algorithm starts from an independent set S, and replaces S by an independent set $S'\in\mathcal{N}_k(S)$ until S is k-optimal. Optimising over the k-opt neighbourhood can be done by trying all possibilities of U and W. There are $\binom{|S|}{k-1}$ possible ways to choose U, and at most $\binom{|V|}{k}$ possible ways to choose W. Checking feasibility takes $O(|E|)$ time. It follows that searching the k-opt neighbourhood can be done in $O(|V|^{2k-1}|E|)$ time.

Note, that in order to compute a 1-optimal solution it is sufficient to look at each node only once, and only checking feasibility on arcs adjacent to this node. Therefore, a 1-optimal solution can be computed in $O(|V|+|E|)$ time.

The following proposition tells that we can take advantage of the sparsity of a graph when looking for neighbours in the k-opt neighbourhood of a $(k-1)$-optimal independent set S.

Proposition 4.6. *Let S be a $(k-1)$-optimal independent set for some $k > 1$, and $S' \in \mathcal{N}_k(S)$. Then $S' = (S \setminus U) \cup W$ for some sets $U \subseteq S$ and $W \subseteq N(U)$ such that $G[N_1(U)]$ is connected.*

Proof. Suppose by way of contradiction that $W \not\subseteq N(U)$. Then, there exists $v \in W \setminus N_1(U)$, and because $S' \in \mathcal{N}_k(S)$ we have that $S \cup \{v\}$ is an independent set, so S is not 1-optimal, contradicting our choice of S. It follows that $W \subseteq N(U)$.

From the $(k-1)$-optimality of S it follows that $|U| = k-1$. Now, let $X \subseteq N_1(U)$ be the node set of a connected component in $G[N_1(U)]$ with $|(U \cap X)| = |(W \cap X)| - 1$. Note that such a node set exists because $|S'| = |S| + 1$. Then, the set $I = (S \setminus (U \cap X)) \cup (W \cap X)$ is an independent set with $|I| = |S|+1$. It follows from the $(k-1)$-optimality of S that $|U \cap X| = k-1$. But then, $U \cap X = U$, so $U \subseteq X$, which implies that $G[N_1(U)]$ is connected. □

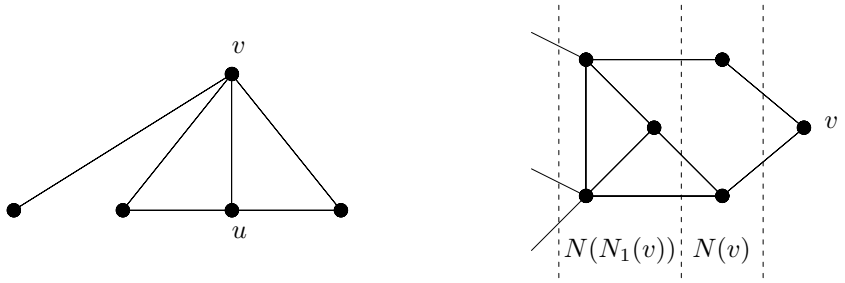
So, when searching the k-opt neighbourhood of S we can limit our choice of U and W using the above observations. This results in a more efficient search of the k-opt neighbourhood on sparse graphs with a large diameter.

4.3 A Branch-and-Cut Algorithm

Recall the definitions of $P_E, P_{\mathcal{C}}, P_{\mathcal{C}}^{\text{GUB}}$, and P_{IS} from Section 4.1.2. To solve independent set problems, we use the standard branch-and-cut algorithm of Section 3.3 starting with $P_{\mathcal{C}}^{\text{GUB}}$ in the root node for some collection $\mathcal{C}$ of maximal cliques that covers all edges in E. The formulation is improved by applying some pre-processing, which is discussed in Section 4.3.1. We use the following valid inequalities for P_{IS} to strengthen our formulation of the maximum independent set problem: maximal clique inequalities, lifted odd hole inequalities, and mod-k inequalities. These are the subjects of Sections 4.3.2, 4.3.3, and 4.3.4, respectively. In addition, we use the strengthened conditions for setting variables based on reduced cost from Section 3.2.3. Whenever we set variables either by reduced cost or by branching, we try to set more variables by using logical implications. These are described in Section 4.3.5. In each node of the branch-and-bound tree, we may use the rounding algorithms of Section 4.2 to find integer solutions.

4.3.1 Preprocessing Techniques

In this section we consider ways to recognise nodes that belong to a maximum independent set, or that do not belong to any maximum independent set at all. Nodes that belong to a maximum independent set can be set to one, and nodes that do not belong to any maximum independent set can be set to zero before starting the branch-and-cut algorithm.



v is node-dominated by u v is set-dominated by $N(v)$

Figure 4.1: Node- and Set-Dominance Criteria

The following result allows us to identify nodes in a graph that belong to a maximum independent set. A weighted version was already mentioned by Nemhauser and Trotter [87, Theorem 1].

Proposition 4.7. *If $I \subseteq V$ is a maximum independent set in $G[N_1(I)]$, then there exists a maximum independent set in G that contains I.*

Proof. Suppose that $I \subseteq V$ is a maximum independent set in $G[N_1(I)]$. Let I^* be a maximum independent set in G, $I_1 = I^* \setminus N_1(I)$, and $I_2 = I_1 \cup I$. By construction, I_2 is an independent set. Observe that

$$|I_1| = |I^* \setminus N_1(I)| = |I^*| - |I^* \cap N_1(I)| \geq |I^*| - |I|$$

because $I^* \cap N_1(I)$ is an independent set in $G[N_1(I)]$. Hence

$$|I_2| = |I_1 \cup I| = |I_1| + |I| \geq |I^*|.$$

So I_2 is a maximum independent set that contains I. □

A *simplicial node* is a node whose neighbours form a clique. As a corollary to Proposition 4.7 we have that we can set x_v to one if v is a simplicial node. Checking whether a given node v is simplicial can be done by marking its neighbours, and then for each neighbour of v, counting whether it is adjacent to all other neighbours of v. This check takes at most $O(|\delta(v)| + \sum_{u \in N(v)} |\delta(u)|)$ for node v, summing up to a total of $\sum_{v \in V} O(|\delta(v)|^2) \leq O(|V||E|)$ time for all nodes.

The following two propositions, illustrated in Figure 4.1, are special cases of the dominance criteria reported by Zwaneveld, Kroon and van Hoesel [129].

Proposition 4.8. (Node-Dominance) *Let $u, v \in V$ be nodes in G such that $N_1(u) \subseteq N_1(v)$. Then there exists a maximum independent set in G that does not contain v.*

Proof. Let I^* be a maximum independent set. If $v \notin I^*$, we are done. Otherwise, let $I_1 = I^* \setminus \{v\}$, and $I_2 = I_1 \cup \{u\}$. Because $v \in I^*$ and $N_1(u) \subseteq N_1(v)$,

$I_1 \cap N_1(u) = \varnothing$. It follows that I_2 is an independent set with $|I_2| = |I^*|$, hence it is maximum and $v \notin I_2$. □

If the condition of Proposition 4.8 is satisfied, we say that v is *node-dominated* by u. Searching for a node u that node-dominates v can be done in a similar way as testing whether v is simplicial. As a corollary to Proposition 4.8, we can set x_v to zero if v is node-dominated.

Proposition 4.9. (Set-Dominance) *Let $v \in V$ be a node in G. If for each independent set I in $G[N(N_1(v))]$ there exists a node $u \in N(v)$ with $N(u) \cap I = \varnothing$, then there exists a maximum independent set in G that does not contain v.*

Proof. Let I^* be a maximum independent set. If $v \notin I^*$, we are done. Otherwise, let $I_1 = I^* \setminus \{v\}$, $u \in N(v)$ a node with $N(u) \cap I_1 = \varnothing$, and $I_2 = I_1 \cup \{u\}$. It follows directly from the choice of u that I_2 is an independent set with $|I_2| = |I^*|$ that satisfies the requirement. □

If the condition of Proposition 4.9 is satisfied, we say that v is *set-dominated* by $N(v)$. Zwaneveld, Kroon and van Hoesel presented weighted versions of the node-dominance and set-dominance criteria.

We now focus our attention on how to determine whether a node is set-dominated.

Proposition 4.10. *Let $v \in V$. There exists a node $u \in N(v)$ with $N(u) \cap I = \varnothing$ for each independent set I in $G[N(N_1(v))]$ if and only if there does not exist an independent set I in $G[N(N_1(v))]$ with $N(v) \subseteq N(I)$.*

There does not seem to be a simple way to determine whether a node is set-dominated or not. Instead, we use a recursive algorithm, by Zwaneveld *et al.*, that searches for an independent set I in $G[N(N_1(v))]$ with $N(v) \subseteq N(I)$. Although the algorithm has an exponential worst-case time bound, it turns out to work efficiently on several classes of problem instances that are of interest.

The algorithm has as input two sets of nodes, denoted by U and W, and returns true if and only if there exists an independent set in $G[W]$ that has as neighbours all nodes in U. If U is empty the algorithm returns true. If U is not empty but W is, the algorithm returns false. Otherwise it selects a node $v \in U$. For each $w \in W \cap N_1(v)$, the algorithm recursively searches for an independent set in $G[W \setminus N_1(w)]$ that has as neighbours all nodes in $U \setminus N_1(w)$. If any of the recursive calls returns true, the algorithm returns true; otherwise it returns false. If the algorithm returns false when applied to $U = N(v)$, and $W = N(N_1(v))$, then v is set-dominated.

The correctness of the algorithm can be shown by induction, where the induction step follows from the observation that if there exists an independent set in $G[W \setminus N_1(w)]$ that has as neighbours all nodes in $U \setminus N_1(w)$, then we can extend it with node w to become an independent set in W that has as neighbours all nodes in U. As a corollary to Proposition 4.9 we can set x_v to zero if v is set-dominated.

4.3.2 Maximal Clique inequalities

Let $\boldsymbol{x} \in P_E$ be a fractional solution to the maximum independent set problem. Our procedure to identify violated maximal clique inequalities is a combination of two greedy heuristics described by Nemhauser and Sigismondi [85].

We look for a violated maximal clique inequality starting from each node v in the fractional support from $\boldsymbol{x}$. For any node v this is done as follows. We initialise $C := \{v\}$, and set $N := N_1(v) \setminus v$. We maintain as invariant that N contains all nodes in G that are adjacent to each node in C. As long as N is not empty, we extract a node u from N, add this node to C, and remove all nodes from N that are not adjacent to u. We use the following two criteria to choose u, each defining its own greedy heuristic:

$$u = \arg\min_{u \in N}\{|x_u - \tfrac{1}{2}|\}, \qquad \text{and} \qquad u = \arg\max_{u \in N}\{x_u\}.$$

When upon termination N is empty, C is a maximal clique by our invariant.

Because $|N| \leq \delta(v)$, the procedure that looks for a maximal clique starting from a given node can be implemented in $O(|\delta(v)|^2)$ (using an incidence matrix to determine in $O(1)$ time whether two nodes in G are adjacent). As a consequence, the whole procedure can be made to work in time

$$\sum_{v \in V} O(|\delta(v)|^2) \leq |V| \cdot \sum_{v \in V} O(\delta(v)) = O(|V||E|).$$

4.3.3 Lifted Odd Hole Inequalities

Let $\boldsymbol{x} \in P_E$ be a fractional solution. A *hole* in G is a cycle G without chords. An *odd hole* in G is a hole in G that contains an odd number of nodes. If H is an odd hole in G, then the *odd hole inequality* defined by H is

$$\boldsymbol{x}(V(H)) \leq \lfloor |V(H)|/2 \rfloor.$$

It was shown by Padberg [90] that the odd hole inequality is valid for P_{IS}, and facet-defining for $P_{\text{IS}} \cap \{\boldsymbol{x} \mid \boldsymbol{x}_{V \setminus V(H)} = \boldsymbol{0}\}$. Given an odd hole H, a *lifted* odd hole inequality is of the form

$$\boldsymbol{x}(V(H)) + \sum_{v \in V \setminus V(H)} \alpha_v x_v \leq \lfloor |V(H)|/2 \rfloor \tag{4.9}$$

for some suitable vector $\boldsymbol{\alpha} \in \mathbb{R}^V$. We compute values $\boldsymbol{\alpha} \in \mathbb{N}^V$ using sequential lifting (Theorem 3.3), to obtain facet-defining inequalities of P_{IS}.

The separation algorithm for lifted odd hole inequalities consists of two parts. The first part derives an odd hole H from $\boldsymbol{x}$ that defines a violated or nearly violated odd hole inequality. The second part consists of lifting the resulting odd hole inequality so that it becomes facet-defining for P_{IS}. After the lifting, we check whether we have found a violated inequality and if so, we report it.

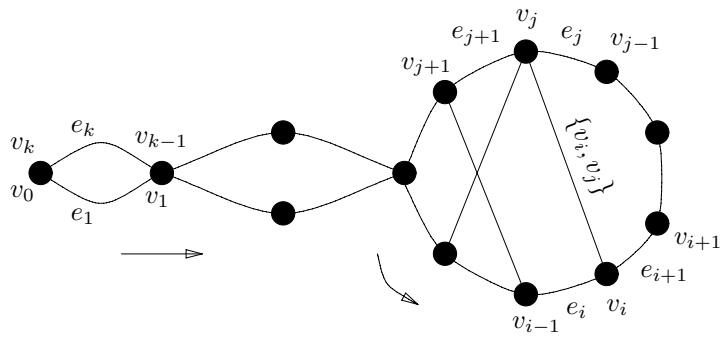


Figure 4.2: Identifying an odd hole in a closed walk.

Identifying a (Nearly) Violated Odd Hole Inequality

We start by describing the separation algorithm for the basic odd hole inequalities $\boldsymbol{x}(V(H)) \leq \lfloor |V(H)|/2 \rfloor$ given the vector $\boldsymbol{x}$. We first find an odd cycle starting from some node $v \in V$ using the construction described by Grötschel, Lovász, and Schrijver [55]. To find a shortest odd cycle containing node v, Grötschel *et al.* construct an auxiliary bipartite graph $\tilde{G} = ((V^1, V^2), \tilde{E})$ and cost vectors $\boldsymbol{c} \in [0,1]^E$ and $\tilde{\boldsymbol{c}} \in [0,1]^{\tilde{E}}$ as follows. Each node $v \in V$ is split into two nodes v^1 and v^2, with v^i included in V^i $(i = 1, 2)$. For each edge $\{u, v\} \in E$, we add the edges $\{u^1, v^2\}$ and $\{u^2, v^1\}$ to $\tilde{E}$, and set $c_{\{u,v\}} = \tilde{c}_{\{u^1,v^2\}} = \tilde{c}_{\{u^2,v^1\}} = 1 - x_u - x_v$. Observe that a path from $u^1 \in V^1$ to $v^2 \in V^2$ in $\tilde{G}$ corresponds to a walk of odd length in G from u to v.

A shortest path from v^1 to v^2 in $\tilde{G}$ corresponds to a shortest odd length closed walk in G containing v. The reason that we are looking for a shortest path is that an odd hole H in G defines a violated odd hole inequality if $\boldsymbol{c}(E(H)) < 1$, and a short closed walk in $\tilde{G}$ is more likely to lead to a violated lifted odd hole inequality than a long one. In our implementation, we restricted ourselves to shortest paths with length at most 1.125. Shortest paths in a graph with nonnegative edge lengths can be found using Dijkstra's algorithm [42]. Hence, we can find a closed walk

$$C := (v = v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k = v)$$

in G with odd k that is minimal with respect to $\boldsymbol{c}$ and $|C|$ by using Dijkstra's algorithm to find a shortest path (with respect to $\tilde{\boldsymbol{c}}$) of minimal cardinality from v^1 to v^2 in $\tilde{G}$. Some of the v_i may occur more than once, and the walk may have chords.

Let $j \in 2, \ldots, k-1$ be the smallest index such that there exists an edge $\{v_i, v_j\} \in E$ for some $i \in 0, \ldots, j-2$ (such i, j exist because $\{v_0, v_{k-1}\} \in E$). Let $i \in 0, \ldots, j-2$ be the largest index such that $\{v_i, v_j\} \in E$. Let

$$H := (v_i, e_{i+1}, v_{i+1}, \ldots, e_j, v_j, \{v_j, v_i\}, v_i).$$

The construction of H is illustrated in Figure 4.2. We proceed by showing that H is indeed an odd hole.

Proposition 4.11. *Let H be constructed as above. Then, H is an odd hole in G.*

Proof. Because $\{v^1, v^2\} \notin \tilde{E}$, we have that $|H| \geq 3$. Clearly H is a cycle in G. By choice of i and j, H does not contain chords, so H is a hole in G. It remains to prove that $|V(H)|$ $(= j - i + 1)$ is odd. Suppose by way of contradiction that $|V(H)|$ is even. Then,

$$C' := (v = v_0, e_1, \ldots, v_{i-1}, e_i, v_i, \{v_i, v_j\}, v_j, e_{j+1}, v_{j+1}, \ldots, e_k, v_k = v)$$

is an odd length closed walk in G containing v. Moreover,

$$\boldsymbol{c}(\{e_{i+1}, \ldots, e_j\}) = (j - i) - (2\textstyle\sum_{p=i+1}^{j-1} x_{v_p}) - x_{v_i} - x_{v_j}.$$

It follows from $\boldsymbol{x}(\{v_p, v_{p+1}\}) \leq 1$ that

$$\sum_{p=i+1}^{j-1} x_{v_p} \leq (j - i - 1)/2.$$

Therefore,

$$\boldsymbol{c}(\{e_{i+1}, \ldots, e_j\}) \geq (j - i) - (j - i - 1) - x_{v_i} - x_{v_j} = c_{\{v_i, v_j\}},$$

so C' is not longer than C with respect to $\boldsymbol{c}$. However, C' is of smaller cardinality, which contradicts our choice of C. Hence H is an odd hole in G. □

If $|V(H)| = 3$, then H is a clique in G, and we ignore it in our computations.

Lifting an Odd Hole Inequality

Let H be an odd hole in G. Assume that we have an ordering of the node set $V \setminus V(H)$ that is given by $\{v_1, v_2, \ldots, v_{|V \setminus V(H)|}\}$. By Theorem 3.3 a lifted odd hole induces a facet if we choose

$$\alpha_{v_i} = \lfloor |V(H)|/2 \rfloor - \max\{\boldsymbol{x}(V(H)) + \textstyle\sum_{j=1}^{i-1} \alpha_{v_j} x_{v_j} \mid \boldsymbol{x} \in X^i_{\mathrm{IS}}\},$$

where

$$X^i_{\mathrm{IS}} = \{\chi^I \mid I \text{ is an independent set in } G[(V(H) \cup \{v_1, \ldots, v_{i-1}\}) \setminus N_1(v_i)]\}.$$

In order to compute the lifting coefficients, we have to compute several maximum weight independent sets, one for each lifting coefficient.

Nemhauser and Sigismondi [85] observed that $\alpha_v = 0$ for $v \in V \setminus V(H)$ if $|N_1(v) \cap V(H)| \leq 2$. This implies that the independent set problems that have to be solved in order to compute the lifting coefficients $\boldsymbol{\alpha}$ are relatively small in practice. We lift the variables in non-decreasing lexicographic order of the pairs $(|\frac{1}{2} - x_v|, -|N_1(v) \cap V(H)|)$, where ties are broken at random.

To compute the coefficients $\boldsymbol{\alpha}$, we will make use of a path decomposition (see Bodlaender [19], and de Fluiter [46]) of the graph induced by the nodes in the hole and the nodes we already lifted.

Definition 4.3. A *path decomposition* of a graph $G = (V, E)$ is a sequence $(S_i)_{i=1}^n$ satisfying the following conditions:

$$\bigcup_{i=1}^n S_i = V, \tag{4.10}$$

$$\text{for all } \{u, v\} \in E \text{ there exists } i \in \{1, \dots, n\} \text{ with } \{u, v\} \subseteq S_i, \text{ and} \tag{4.11}$$

$$\text{for all } i, j, k \text{ with } 1 \leq i < j < k \leq n \text{ we have } S_i \cap S_k \subseteq S_j. \tag{4.12}$$

The *width* of a path decomposition $(S_i)_{i=1}^n$ is the value $\max_{i=1}^n |S_i| - 1$.

We may assume without loss of generality that S_i and S_{i+1} differ in only one node, i.e., that $S_{i+1} = S_i \cup \{v\}$ or $S_{i+1} = S_i \setminus \{v\}$ for some $v \in V$. We may also assume without loss of generality that $S_1 = S_n = \varnothing$. A path decomposition satisfying these assumptions is called *normalised*. A normalised path decomposition has $n = 2|V| + 1$. We will present our algorithm to compute the maximum weight of an independent set given a path decomposition in the next sub-section. Here, we proceed by outlining how we obtain and maintain the path decomposition that we use for this purpose.

Given a hole $H = (v_0, e_1, v_1, \dots, e_n, v_n = v_0)$, an initial path decomposition of the graph $G[V(H)] = (V(H), E(H))$ of width two is given by

$$S_i = \begin{cases} \varnothing, & \text{if } i = 1 \text{ or } i = 2|V(H)| + 1, \\ \{v_0\}, & \text{if } i = 2 \text{ or } i = 2|V(H)|, \\ \{v_0, v_k\}, & \text{if } 2 < i < 2|V(H)| \text{ and } i = 2k + 1, \\ \{v_0, v_k, v_{k+1}\}, & \text{if } 2 < i < 2|V(H)| \text{ and } i = 2(k + 1). \end{cases}$$

Suppose at some stage we want to compute the lifting coefficient for some node $v \in V$. Let V' be the set of nodes that are either in the hole or did already receive a positive lifting coefficient at some earlier stage. Assume that $(S_i)_{i=1}^{2|V'|+1}$ is a path decomposition of the graph induced by V'. A path decomposition of $G[V' \setminus N_1(v)]$ can be obtained from $(S_i)_{i=1}^{2|V'|+1}$ by eliminating the nodes in $N_1(v)$ from all sets S_i $(i = 1, \dots, 2|V'| + 1)$ and eliminating consecutive doubles (i.e., sets S_i and S_{i+1} that are equal).

For each node that we assign a positive lifting coefficient, we have to update our path decomposition. Suppose at some stage we have found a strictly positive lifting coefficient for some node $v \in V$. Let $V', (S_i)_{i=1}^{2|V'|+1}$ be as before. We have to extend the path decomposition so that it becomes a path decomposition for $G[V' \cup \{v\}]$. We do this in a greedy fashion, by identifying the indices j, k such that $j = \min\{i \mid \{u, v\} \in E, u \in S_i\}$ and $k = \max\{i \mid \{u, v\} \in E, u \in S_i\}$, and adding v to all sets S_i for $i \in \{j, \dots, k\}$. Having done this, our path decomposition satisfies conditions (4.10)–(4.12) for the graph $G[V' \cup \{v\}]$. We normalise the resulting path decomposition to ensure that it satisfies our assumptions on the differences between consecutive sets.

Weighted Independent Sets by Path Decomposition

Suppose we are given a graph $G = (V, E)$, together with a normalised path decomposition $(S_i)_{i=1}^{2|V|+1}$ of G. Let $\mathcal{I}$ again denote the collection of all inde-

pendent sets in G, and let $\mathcal{I}_i = \{I \in \mathcal{I} \mid I \subseteq S_i\}$ be all independent sets in $G[S_i]$. Finally, let $V_i = \bigcup_{j=1}^{i} S_j$. Since the path decomposition $(S_i)_{i=1}^{2|V|+1}$ is normalised, for all $i > 1$ there is a node $v \in V$ such that either $S_i = S_{i-1} \cup \{v\}$ or $S_i = S_{i-1} \setminus \{v\}$. As a consequence the sets $\mathcal{I}_i$ satisfy the following recursive relation:

$$\mathcal{I}_i = \begin{cases} \{\varnothing\}, & \text{if } i = 1, \\ \mathcal{I}_{i-1} \cup \{(I \cup \{v\}) \in \mathcal{I} \mid I \in \mathcal{I}_{i-1}\}, & \text{if } i > 1 \text{ and } S_i = S_{i-1} \cup \{v\}, \text{ and} \\ \{I \in \mathcal{I}_{i-1} \mid v \notin I\} & \text{if } i > 1 \text{ and } S_i = S_{i-1} \setminus \{v\}. \end{cases} \tag{4.13}$$

Given a path decomposition $(S_i)_{i=1}^{2|V|+1}$ of G and a weight vector $\boldsymbol{\alpha} \in \mathbb{N}^V$, we can compute the weight of a maximum weight independent set in G with respect to $\boldsymbol{\alpha}$ using a dynamic programming-like algorithm. To describe this algorithm, we use the functions

$$z_i : \mathcal{I}_i \to \mathbb{N} : I \mapsto \max\{\boldsymbol{\alpha}(I') \mid I' \text{ is independent set in } G[V_i],\ I' \cap S_i = I\}.$$

It follows from condition (4.10) that the value of $z_{2|V|+1}(\varnothing)$ is the maximum weight of an independent set of G, which is what we want to compute. From the definition of z_i we find $z_1(\varnothing) = 0$. Now suppose $i > 1$, let $I \in \mathcal{I}$ be an independent set in S_i, and let I^* be the maximum weight independent set in $G[V_i]$ with $I^* \cap S_i = I$. In the following, we will relate I^* to a maximum weight independent set I' in $G[V_{i-1}]$ and characterise $I' \cap S_{i-1}$. There are four cases to consider.

Case (i): $S_i = S_{i-1} \cup \{v\}$ and $v \notin I$. Clearly, I^* also is a maximum weight independent set in $G[V_{i-1}]$ with $I^* \cap S_{i-1} = I$. Hence we can take $I' = I^*$, and $I' \cap S_{i-1} = I$.

Case (ii): $S_i = S_{i-1} \cup \{v\}$ and $v \in I$. We claim that $u \in S_i$ for all $\{u, v\} \in E(V_i)$. It follows from this claim that $I' = I^* \setminus \{v\}$ is a maximum weight independent set in $G[V_{i-1}]$ (or I^* would not be optimal either), and $I' \cap S_{i-1} = I \setminus \{v\}$. To prove our claim suppose by way of contradiction that $\{u, v\} \in E(V_i)$ but $u \notin S_i$. From this it follows that $u \in S_k$ for some $k < i$. By condition (4.11) there exists $j \in \{1, \ldots, 2|V| + 1\}$ such that $\{u, v\} \subseteq S_j$. Since $S_i = S_{i-1} \cup \{v\}$ and by condition (4.12) v occurs only in consecutive sets in the path decomposition, we find that $j > i$. However, condition (4.12) together with $u \in S_k$ and $u \in S_j$ implies that $u \in S_i$, a contradiction, completing the proof of the claim.

Case (iii): $S_i = S_{i-1} \setminus \{v\}$ and $I \cup \{v\}$ is an independent set in $G[S_{i-1}]$. From $V_i = V_{i-1}$ it follows that $I' = I^*$ is a maximum weight independent set in $G[V_{i-1}]$ with $I' \cap S_i = I$. Either $v \in I'$ or $v \notin I'$. Hence, $I' \cap S_{i-1} = I \cup \{v\}$ or $I' \cap S_{i-1} = I$.

Case (iv): $S_i = S_{i-1} \setminus \{v\}$ and $I \cup \{v\}$ is not an independent set in $G[S_{i-1}]$. Since $I^* \cap S_i = I$ implies $(I^* \cup \{v\}) \cap S_{i-1} = I \cup \{v\}$, and I^* is an independent set, we have $v \notin I^*$. It follows that we can take $I' = I^*$, and $I' \cap S_{i-1} = I$.

The case analyses above leads to the following recurrence for $z_i(I)$ with $i > 1$:

$$z_i(I) = \begin{cases} z_{i-1}(I) & \text{if } S_i = S_{i-1} \cup \{v\} \text{ and } v \notin I, \\ z_{i-1}(I \setminus \{v\}) + \alpha_v, & \text{if } S_i = S_{i-1} \cup \{v\} \text{ and } v \in I, \\ \max(z_{i-1}(I), z_{i-1}(I \cup \{v\})), & \text{if } S_i = S_{i-1} \setminus \{v\} \text{ and } I \cup \{v\} \in \mathcal{I}_{i-1}, \\ z_{i-1}(I), & \text{if } S_i = S_{i-1} \setminus \{v\} \text{ and } I \cup \{v\} \notin \mathcal{I}_{i-1}. \end{cases} \tag{4.14}$$

The dynamic programming algorithm works in $2|V| + 1$ iterations, numbered $1, \ldots, 2|V| + 1$. In iteration i, the algorithm computes the domain of z_i, together with its function values. The domain of z_1 is $\{\varnothing\}$, and $z_1(\varnothing) = 0$. For $i > 1$, the domains and function values are computed unfolding the recursive definitions (4.13) and (4.14), respectively.

To implement the dynamic programming algorithm, we maintain a list $\mathcal{L}$ of tuples $(I, z_i(I))$ with $I \in \mathcal{I}_i$, where i is the iteration number. Iteration 1 is implemented by setting $\mathcal{L} = \{(\varnothing, 0)\}$. Iteration $i > 1$ is implemented as follows. Either $S_i = S_{i-1} \cup \{v\}$ or $S_i = S_{i-1} \setminus \{v\}$. If $S_i = S_{i-1} \cup \{v\}$ we update $\mathcal{L}$ by adding the tuple $(I \cup \{v\}, z + \alpha_v)$ to $\mathcal{L}$ for each tuple $(I, z) \in \mathcal{L}_{i-1}$ that satisfies $N_1(v) \cap I = \varnothing$. If $S_i = S_{i-1} \setminus \{v\}$, we update $\mathcal{L}$ by removing all tuples (I, z) that have $v \in I$. After removing a tuple (I, z) from $\mathcal{L}$, we replace the tuple $(I \setminus \{v\}, z') \in \mathcal{L}$ by the tuple $(I \setminus \{v\}, \max(z, z'))$.

If we order $\mathcal{L}$ such that for all pairs of tuples $(I_1, z_i(I_1)), (I_2, z_i(I_2)) \in \mathcal{L}$ with $I_2 \subset I_1$ we have that the tuple containing I_1 precedes the tuple containing I_2 in $\mathcal{L}$, each iteration can be implemented by a single pass over $\mathcal{L}$, using two pointers into $\mathcal{L}$. If the path decomposition $(S_i)_{i=1}^{2|V|+1}$ has width w, then the independent sets in the domains of each of the functions z_i can be implemented using bit vectors of length w, and the ordering can be implemented using the default "greater than" comparison operator on bit vectors. If we restrict ourselves to path decompositions with width at most 32, then bit vectors can be implemented using integers. In that case the dynamic programming algorithm can be implemented using a double-linked list of tuples of integers as most complicated data structure.

Proposition 4.12. *A maximum weight independent set can be computed using a path decomposition of width w in $O(w2^w|V| + |E|)$ time.*

Proof. When adding a node v, we have to check whether $I \cup \{v\} \in \mathcal{I}$ for each $(I, z) \in \mathcal{L}$. This can be done in constant time for each such I after we mark the neighbours of node v. Making a new independent set takes at most $O(w)$ time. It follows that iteration i in which a node is added can be implemented in $O(w|\mathcal{I}_i| + |\delta(v)|)$, where v is the node we add. If we delete a node in iteration

i, all checks and updates can be done in constant time, so the iteration can be implemented in $O(|\mathcal{I}_i|)$ time. Since $|\mathcal{I}_i| \leq 2^w$ for all i, the dynamic programming algorithm can be implemented to work in

$$\sum_{i=1}^{2|V|+1} O(w2^w + |\delta(v)|) = O(w2^w|V| + |E|)$$

time. □

The factor 2^w in the time complexity of the dynamic programming algorithm looks rather ominous. However, the actual running time depends on the actual size that $\mathcal{L}$ attains. In iteration i, this size equals the number of independent sets in $G[S_i]$. In our implementation, we restricted ourselves to path decompositions with width at most 24, and it rarely occurred that the path width exceeded 24. On those rare occasions, however, we do not compute the true lifting coefficients but use a coefficient of 0 instead. Counting to 2^{24} each time we want to compute a lifting coefficient would definitely stall our algorithm. Although we cannot exclude the possibility that this occurs, we would like to stress that this did not occur in our computational experiments. We see this as an indication that the worse case behaviour is unlikely to manifest itself on the classes of problem instances that we are interested in.

4.3.4 Maximally Violated Mod-k Cuts

Suppose $\boldsymbol{x}^* \in P_E$ is a fractional solution to the maximum independent set problem. We use the algorithm of Section 3.3.5 to search maximally violated mod-k cuts. As input to the mod-k separation algorithm, we use all (globally) valid inequalities for P_{IS} that are present in the formulation of the linear programming relaxation and are satisfied with equality by $\boldsymbol{x}^*$. The inequalities we use are the following: maximal clique inequalities (4.2), lifted odd hole inequalities (4.9), non-negativity constraints on $\boldsymbol{x}_V$, upper bound constraints of 1 on components of $\boldsymbol{x}_V$, and mod-k cuts that were found at an earlier stage of the algorithm.

4.3.5 Logical Implications

In each node of the branch-and-bound tree we solve a linear programming relaxation of the form

$$z^* = \max\{z(\boldsymbol{x}) = \boldsymbol{x}(V) \mid A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\}, \tag{4.15}$$

for some $\boldsymbol{l}, \boldsymbol{u} \in \{0,1\}^V$ where $A, \boldsymbol{b}$ are obtained from the constraint matrix of $P_{\mathcal{C}}^{\text{GUB}}$ and $\mathbf{1}$, respectively, by adding the rows and right hand sides of valid inequalities that are produced by our separation algorithms. Recall the notion of setting variables from Section 3.2.2.

We start with the most elementary of all logical implications:

Proposition 4.13. *Let $v \in V$ be a node in G and let $W = N(v)$ be its set of neighbours. If x_v is set to one, then x_w can be set to zero for all $w \in W$ without changing z^*.*

Proof. Directly from the definition of P_E. □

Proposition 4.14. *Let v, W be as in Proposition 4.13. If x_w is set to zero for all $w \in W$, then x_v can be set to one without changing the value of z^*.*

Proof. Because $z^* = z(\boldsymbol{x}^*)$ for some optimal solution $\boldsymbol{x}^*$ to model (4.15), and $\boldsymbol{x}^*_W = \mathbf{0}$ together with optimality of $\boldsymbol{x}^*$ implies that $x^*_v = 1$, the proposition holds. □

If we set x_v to one for some $v \in V$, then we can set x_w to zero for all neighbours w of v by Proposition 4.13. In a formulation of the independent set problem with explicit slack variables, such as model (4.4), it is possible to interpret the slack variable s_C that is associated with clique $C \in \mathcal{C}$ as a variable that is associated with an extra node that is connected to all nodes in C. Using this interpretation we can also apply Proposition 4.13 to the slack variables s_C for all $C \in \mathcal{C}$. If we set x_v to zero for some $v \in V$, its neighbours may satisfy the conditions of Proposition 4.14.

After applying Proposition 4.13 to all nodes that are set to one, we can restate the linear programming problem 4.15 in terms of its free variables. The resulting linear programming problem can be interpreted as a (fractional) independent set problem in the graph induced by the nodes that correspond to the free variables (i.e., with a lower bound of $\mathbf{0}$ and an upper bound of $\mathbf{1}$). Therefore, setting variables to zero can be interpreted as removing nodes from the graph.

Proposition 4.15. *Let $v \in G$, $U = N(v)$ the set of neighbours of v, and $W = N(N_1(v))$ the set of neighbours of the 1-neighbourhood of v. When removing v from G, the following cases can occur:*

(i) the nodes in U may become simplicial nodes,

(ii) the nodes in W may become node-dominated by nodes in U, or

(iii) the nodes in W may become set-dominated.

Proof. Directly from Propositions 4.7, 4.8, and 4.9. □

After setting a variable to zero, we check whether any of the three cases of Proposition 4.15 occurs, and if so, we take the appropriate action.

4.3.6 Branching Scheme

We complete the description of our branch-and-cut algorithm by giving the branching scheme we use. Since we use the formulation of the maximum independent set problem with explicit slack variables (4.4) all maximal clique

Name	Heuristic	Section
O1	1-Opt starting from zero	4.2.3
O2	1-Opt followed by 2-opt starting from zero	4.2.3
L1	Simple LP rounding followed by 1-opt	4.2.1, 4.2.3
L2	Simple LP rounding followed by 1-opt and 2-opt	4.2.1, 4.2.3
R1	Minimum regret rounding with parameter $t = 1$	4.2.2
R2	Minimum regret rounding with parameter $t = 2$	4.2.2

Table 4.1: Heuristics for the Maximum Independent Set Problem

inequalities are GUB constraints. We try to take advantage of this by using branching on variables and branching on GUB constraints. Deciding whether to branch on a variable or a GUB constraint is done based on pseudo-cost as described in Section 3.2.5.

4.4 Computational Results

In this section we report on the computational behaviour of the algorithms proposed in this chapter, which we implemented using the graph data structure from LEDA [83]. We tested our algorithms on independent set problems that are similar to those reported on in the literature. An interesting class of independent set problem instances that can be found in the literature is the class defined on so-called *uniform random graphs*, or *URGs*. URGs are graphs on n nodes in which each of the $n(n-1)/2$ possible arcs is present with a fixed probability p. All averages reported in this section are taken over 25 randomly generated instances for every choice of n and p. We will first consider the performance of our heuristic algorithms, and then proceed by discussing the behaviour of our optimisation algorithms.

Heuristics. The heuristics for the maximum independent set problem presented in Section 4.2 are summarised in Table 4.1. The average sizes of the maximum independent sets computed by our heuristics and the corresponding running times are reported on in Table 4.2. The columns O1 and O2 refer to the iterative improvement algorithms that were presented in Section 4.2.3. These algorithms use the 1-opt and 2-opt neighbourhoods, respectively, starting from $\mathbf{0}$. The L1, L2, R1 and R2 columns all refer to LP-based rounding algorithms that first compute the optimal solution to an LP relaxation of the independent set polytope that is strengthened by our cutting plane routines. The L1 and L2 columns refer to the algorithms that first apply the simple LP rounding algorithms from Section 4.2.1 to obtain an integer solution. Next, they invoke an iterative improvement algorithm starting from this integer solution using the 1-opt and 2-opt neighbourhoods, respectively. The R1 and R2 columns refer to algorithms that apply the minimum regret rounding heuristic from Section 4.2.2 to the optimal solution of the LP relaxation, with the parameter t equal to 1

		O1		O2		L1		L2		R1		R2		
n	p	α	CPU	α	CPU	α	CPU	α	CPU	α	CPU	α	CPU	$\alpha(G)$
80	.1	21.5	0.0	24.3	0.0	25.2	0.0	26.3	0.0	26.8	0.0	27.0	0.1	27.3
	.2	13.7	0.0	15.2	0.0	14.5	0.0	15.6	0.0	16.7	0.0	16.8	0.3	18.0
	.3	9.6	0.0	11.2	0.0	9.7	0.0	11.2	0.0	12.7	0.0	12.7	0.4	13.4
	.4	8.0	0.0	8.8	0.0	8.0	0.0	8.8	0.0	9.7	0.0	10.1	0.4	10.6
	.5	6.0	0.0	6.9	0.0	6.0	0.0	6.9	0.0	8.2	0.0	8.3	0.4	8.8
	.6	4.9	0.0	5.6	0.0	4.9	0.0	5.6	0.0	6.7	0.0	6.9	0.4	7.2
	.7	3.8	0.0	4.6	0.0	3.8	0.0	4.6	0.0	5.3	0.0	5.5	0.3	5.9
	.8	3.1	0.0	3.9	0.0	3.2	0.0	4.0	0.0	4.4	0.0	4.5	0.2	4.8
	.9	2.4	0.0	3.0	0.0	3.6	0.0	3.7	0.0	3.7	0.0	3.7	0.0	3.7
90	.1	21.9	0.0	25.2	0.0	25.8	0.0	27.8	0.0	28.4	0.0	28.7	0.2	29.3
	.2	14.3	0.0	16.0	0.0	14.5	0.0	16.2	0.0	18.3	0.0	18.4	0.5	19.1
	.3	9.9	0.0	11.6	0.0	10.0	0.0	11.6	0.0	12.9	0.0	13.1	0.6	14.2
	.4	7.7	0.0	8.8	0.0	7.7	0.0	8.8	0.0	10.2	0.0	10.4	0.6	11.2
	.5	6.4	0.0	7.3	0.0	6.4	0.0	7.3	0.0	8.1	0.0	8.0	0.6	9.1
	.6	5.0	0.0	5.8	0.0	5.0	0.0	5.8	0.0	6.9	0.0	7.0	0.6	7.4
	.7	4.0	0.0	4.8	0.0	4.0	0.0	4.8	0.0	5.6	0.0	5.8	0.5	6.1
	.8	3.2	0.0	3.6	0.0	3.2	0.0	3.6	0.0	4.4	0.0	4.6	0.3	4.8
	.9	2.4	0.0	3.0	0.0	3.8	0.0	3.8	0.0	3.8	0.0	3.8	0.0	3.8
100	.1	23.1	0.0	26.2	0.0	25.8	0.0	27.9	0.0	29.4	0.0	29.5	0.3	30.4
	.2	14.4	0.0	16.3	0.0	14.6	0.0	16.6	0.0	18.3	0.0	18.9	0.7	19.8
	.3	10.4	0.0	12.1	0.0	10.4	0.0	12.2	0.0	13.6	0.0	13.7	0.9	14.7
	.4	8.4	0.0	9.2	0.0	8.4	0.0	9.2	0.0	10.5	0.0	10.8	0.9	11.4
	.5	6.2	0.0	7.3	0.0	6.2	0.0	7.3	0.0	8.4	0.0	8.6	1.0	9.2
	.6	5.1	0.0	6.0	0.0	5.1	0.0	6.0	0.0	6.9	0.0	7.1	0.9	7.6
	.7	4.0	0.0	4.8	0.0	4.0	0.0	4.8	0.0	5.7	0.0	5.6	0.8	6.1
	.8	3.3	0.0	4.1	0.0	3.3	0.0	4.1	0.0	4.4	0.0	4.5	0.6	5.0
	.9	2.6	0.0	3.0	0.0	3.7	0.0	3.7	0.0	3.9	0.0	3.9	0.1	4.0
110	.1	24.4	0.0	28.0	0.0	27.2	0.0	29.6	0.0	31.1	0.0	31.1	0.4	32.3
	.2	14.7	0.0	17.1	0.0	15.0	0.0	17.2	0.0	19.1	0.0	19.1	1.1	20.6
	.3	11.1	0.0	12.0	0.0	11.1	0.0	12.0	0.0	13.7	0.0	13.9	1.3	15.0
	.4	8.5	0.0	9.6	0.0	8.5	0.0	9.6	0.0	10.7	0.0	10.8	1.4	11.7
	.5	6.6	0.0	7.6	0.0	6.6	0.0	7.6	0.0	8.5	0.0	8.9	1.4	9.4
	.6	5.2	0.0	6.0	0.0	5.2	0.0	6.0	0.0	7.0	0.0	7.0	1.3	7.8
	.7	4.4	0.0	5.0	0.0	4.4	0.0	5.0	0.0	5.6	0.0	5.8	1.2	6.1
	.8	3.4	0.0	4.0	0.0	3.4	0.0	4.0	0.0	4.6	0.0	4.8	0.8	5.0
	.9	2.8	0.0	3.1	0.0	2.9	0.0	3.2	0.0	3.7	0.0	3.8	0.2	4.0

Table 4.2: Heuristics on Uniform Random Graphs (URGs)

and 2, respectively. The α columns contain the average size of the resulting independent set, the CPU columns contain the average number of CPU seconds that were needed to find the reported independent sets, the time needed to solve the LP relaxations excluded. Finally, the $\alpha(G)$ column contains the average size of the maximum independent set in the test instances.

On uniform random graphs the 2-opt algorithm consistently performs better than the 1-opt algorithm (on average). Using a rounded LP solution as a starting point of an iterative improvement algorithm improves the quality of the reported independent sets for sparse problems ($p \leq .2$) and for dense problems ($p = .9$). The minimum regret heuristics perform consistently better (on average) than the simple rounding plus iterative improvement heuristics. All rounding algorithms are fast in practice (CPU time $< .1s$ on average, with the exception of minimum regret rounding with $t = 2$ that runs within $1.5s$), the CPU time needed for the rounding phase is dominated by the CPU time needed for solving the linear programs. This makes these algorithms well suited for use from within an LP based branch-and-bound algorithm. Note that the minimum regret heuristic with parameter $t = 2$ often achieves an average value of α that is greater than $\alpha(G) - 1$. This can only happen if the algorithm reports optimal solutions on some of the instances.

Optimisation Algorithms. The average performance of our branch-and-cut and cutting plane/branch-and-bound algorithms are presented in Tables 4.3 and 4.4, respectively. In our cutting plane/branch-and-bound algorithm we solve the root node in exactly the same way as we do in our branch-and-cut algorithm, and then invoke the MIP solver from CPLEX with the strongest possible formulation and the best primal bound we have available. In these tables, the Nodes column contains the average number of nodes in the branch-and-bound tree, and the CPU column contains the average CPU time it took for the algorithm to terminate in seconds. The remaining columns report on the effectiveness of the valid inequalities we use, and their corresponding separation algorithms. For each class of valid inequalities, the # column contains the average number of distinct valid inequalities that were separated, the Gap column contains the average percentage of duality gap closed in the root node, and the Prof. column contain the average amount of CPU time that was spent in the corresponding separation algorithm (Prof. is an abbreviation of *Profiling Data*).

When comparing the branch-and-cut algorithm to the cutting plane/branch-and-bound algorithm, it is clear that the faster of the two algorithms is the cutting plane/branch-and-bound algorithm. We believe that this is due to the fact that we use the MIP solver of CPLEX 6.5 for the branch-and-bound phase, which is a highly optimised commercial package. The branch-and-cut algorithm does achieve the smallest branch-and-bound tree sizes of on almost all of the runs. From the Gap columns it is clear that using our cutting plane routines from each node in the branch-and-bound tree does improve the formulation. Here, maximum clique inequalities become more important when problems are

		Performance		Max Clique			Lifted Odd Hole			Mod-k		
n	p	Nodes	CPU	#	Gap	Prof.	#	Gap	Prof.	#	Gap	Prof.
80	.1	9.2	55.6	221.7	0.3	0.6	220.0	59.0	2.2	121.3	43.8	33.6
	.2	87.3	187.2	425.4	5.1	1.1	327.1	19.7	4.1	423.7	9.1	32.8
	.3	139.6	235.4	702.0	11.8	1.1	92.7	6.3	5.0	364.1	7.6	25.0
	.4	110.2	309.5	997.2	21.0	1.1	22.8	3.2	5.5	333.8	11.8	20.8
	.5	67.8	350.8	1223.3	34.2	1.0	4.7	1.7	3.8	250.8	22.6	17.2
	.6	42.2	378.6	1278.6	48.2	0.8	1.0	0.8	2.4	175.9	33.4	15.6
	.7	18.4	363.8	1302.5	62.2	0.6	0.5	1.3	1.9	114.9	46.7	15.2
	.8	4.5	178.2	1038.8	75.6	0.5	0.0	0.0	1.9	53.6	49.2	15.3
	.9	1.0	15.6	634.4	75.6	1.2	0.0	0.0	5.1	15.2	31.9	50.1
90	.1	16.9	108.7	273.6	0.5	0.6	351.4	51.7	2.5	190.2	33.3	32.6
	.2	272.4	464.6	590.6	5.1	1.4	692.1	20.5	5.4	1033.7	6.4	39.0
	.3	354.5	656.1	1095.9	13.3	1.2	214.2	7.9	5.8	876.6	7.4	27.3
	.4	218.0	748.8	1590.7	21.8	1.2	36.4	3.2	5.6	644.4	12.7	21.4
	.5	131.9	737.2	1794.7	34.4	0.9	3.2	1.0	3.5	382.2	21.2	16.4
	.6	72.3	1023.6	2223.4	47.1	0.7	0.6	0.6	2.2	323.2	33.6	13.5
	.7	30.0	861.2	2002.0	61.4	0.5	0.1	0.5	1.7	184.2	47.1	12.2
	.8	8.0	520.1	1513.3	73.6	0.4	0.1	0.3	1.5	87.6	52.6	14.1
	.9	1.0	29.5	886.5	78.5	1.1	0.0	0.0	5.7	22.2	44.0	53.0
100	.1	65.5	339.2	333.4	0.6	1.0	997.4	41.1	4.5	767.5	23.5	46.7
	.2	766.6	1000.0	844.5	5.7	1.3	992.0	16.9	5.6	1611.5	4.6	33.1
	.3	1008.0	1965.0	1675.2	12.3	1.0	376.9	7.5	5.3	1649.8	6.1	21.5
	.4	712.8	3042.3	2767.4	21.9	1.0	86.6	5.1	4.6	1557.5	11.9	16.7
	.5	244.8	1802.0	2813.0	34.5	0.8	4.8	0.9	3.1	625.2	22.0	14.4
	.6	133.6	2047.8	3085.8	47.1	0.7	1.4	0.5	2.2	473.6	33.4	13.2
	.7	54.8	1749.1	2867.4	58.6	0.5	0.1	0.0	1.9	282.0	43.8	12.2
	.8	7.3	1255.6	2087.6	74.4	0.3	0.0	0.0	1.2	106.7	56.1	8.1
	.9	1.3	183.9	1289.4	86.1	0.5	0.0	0.0	2.6	38.9	59.4	19.4
110	.1	213.6	808.4	395.4	0.8	1.2	2114.0	40.7	5.5	1816.4	21.2	53.8
	.2	2542.4	3309.3	1210.0	5.3	1.0	1883.0	16.5	5.0	3564.9	3.7	26.1
	.3	2528.7	5230.9	2370.5	12.3	0.7	506.7	6.6	3.7	2354.7	5.6	14.2
	.4	1291.7	4984.3	3392.5	22.3	0.7	76.0	3.7	3.4	1510.9	12.2	12.0
	.5	564.9	5564.0	4763.7	34.8	0.7	18.6	3.0	2.6	1301.1	21.7	12.3
	.6	210.4	3935.6	4699.5	46.9	0.7	1.0	0.7	2.1	727.6	32.3	12.7
	.7	82.5	3770.2	4562.8	58.0	0.5	0.2	0.2	1.7	479.7	43.9	12.1
	.8	14.4	2625.8	3174.0	71.6	0.3	0.0	0.0	1.1	181.8	56.8	8.4
	.9	1.6	691.3	1633.1	83.6	0.2	0.0	0.0	1.1	47.0	54.6	7.9

Table 4.3: Run-Time Behaviour of Branch-and-Cut on URGs

		Performance		Max Clique			Lifted Odd Hole			Mod-k		
n	p	Nodes	CPU	#	Gap	Prof.	#	Gap	Prof.	#	Gap	Prof.
80	.1	21.6	26.3	221.1	0.3	0.4	101.3	53.6	1.4	36.5	41.3	18.4
	.2	314.2	34.5	348.0	4.7	0.2	6.9	3.8	0.8	11.5	6.4	4.1
	.3	356.6	59.1	470.2	10.6	0.1	0.2	0.2	0.9	10.1	5.8	2.4
	.4	231.1	95.5	589.1	18.6	0.2	0.0	0.0	1.3	17.1	9.0	3.4
	.5	162.7	136.3	706.3	31.2	0.2	0.0	0.0	0.8	20.2	17.6	3.2
	.6	96.0	179.4	792.6	45.0	0.2	0.0	0.0	0.7	24.4	26.9	3.8
	.7	66.8	256.3	895.9	59.8	0.2	0.0	0.6	0.7	31.1	40.0	4.3
	.8	18.8	226.4	874.1	74.3	0.2	0.0	0.0	0.8	28.0	47.1	4.8
	.9	1.1	59.3	634.3	75.6	0.3	0.0	0.0	1.4	15.2	31.9	13.5
90	.1	75.5	42.3	271.8	0.5	0.3	105.6	42.0	1.2	35.6	27.3	13.7
	.2	954.7	74.8	429.8	4.7	0.1	4.8	2.2	0.6	13.3	4.8	2.5
	.3	754.6	134.9	601.5	11.6	0.1	0.0	0.1	0.9	16.9	6.1	2.1
	.4	402.3	173.2	749.4	19.4	0.2	0.0	0.0	1.0	19.8	10.5	2.6
	.5	200.8	232.6	892.7	31.4	0.2	0.0	0.0	0.6	22.7	16.4	2.8
	.6	117.9	351.4	1044.5	44.1	0.2	0.0	0.0	0.6	28.9	27.1	3.2
	.7	89.5	471.4	1139.5	58.7	0.1	0.0	0.0	0.6	34.3	39.6	3.4
	.8	44.9	506.0	1149.9	72.5	0.1	0.0	0.0	0.7	37.2	49.8	4.3
	.9	1.4	118.2	886.5	78.5	0.3	0.0	0.0	1.4	22.2	44.0	13.6
100	.1	415.2	63.3	327.6	0.6	0.2	96.0	28.1	1.0	32.0	17.5	8.8
	.2	1769.2	146.7	539.0	4.9	0.1	1.9	0.7	0.4	10.4	3.2	1.3
	.3	1426.6	269.2	740.0	10.6	0.1	0.0	0.0	0.8	18.9	4.9	1.5
	.4	679.0	314.6	929.5	19.8	0.1	0.0	0.0	0.8	21.6	9.3	1.9
	.5	323.6	417.5	1118.0	31.2	0.1	0.0	0.0	0.5	26.0	17.4	2.3
	.6	179.1	582.2	1255.2	43.2	0.1	0.0	0.0	0.5	29.3	27.2	2.5
	.7	122.3	798.3	1386.9	55.5	0.1	0.0	0.0	0.6	38.2	36.9	3.6
	.8	70.7	864.6	1427.4	72.5	0.1	0.0	0.0	0.6	35.7	51.6	3.8
	.9	1.3	431.0	1261.3	86.0	0.2	0.0	0.0	1.0	36.0	59.0	7.1
110	.1	901.9	99.2	384.4	0.7	0.2	88.9	23.0	0.7	28.7	14.4	5.1
	.2	3786.0	324.1	641.2	4.2	0.0	0.8	0.2	0.3	10.4	2.3	0.7
	.3	3070.0	496.6	877.5	10.5	0.0	0.0	0.0	0.5	15.1	4.1	0.8
	.4	1505.6	541.4	1130.8	19.6	0.1	0.0	0.0	0.5	21.3	9.2	1.4
	.5	566.1	615.9	1337.4	31.1	0.1	0.0	0.0	0.5	26.2	16.3	2.2
	.6	266.3	777.9	1501.2	42.6	0.1	0.0	0.0	0.5	29.8	24.8	2.5
	.7	149.4	1107.2	1723.9	55.0	0.1	0.0	0.0	0.6	44.6	37.9	3.5
	.8	118.7	1256.0	1790.6	69.6	0.1	0.0	0.0	0.6	38.8	50.7	3.7
	.9	12.0	828.1	1542.6	83.3	0.1	0.0	0.0	0.7	38.4	53.8	4.9

Table 4.4: Run-Time Behaviour of Cutting-Plane/Branch-and-Bound on URGs

more dense, lifted odd hole inequalities become more important for problems with lower densities, and mod-k cuts do well for $p = .1$ and $p \geq .5$.

When comparing the branch-and-bound tree sizes of our branch-and-cut algorithm to those reported in the literature by Mannino and Sassano [80] and Sewell [106], it is clear that we achieve the smallest branch-and-bound tree sizes for very sparse problems ($p \leq .2$) and very dense problems ($p \geq .8$), even though we do not employ a branching scheme that is tailor-made for the independent set problem. We believe that this is due to the quality of the bound obtained from the strengthened LP relaxations that we use. This quality does have its cost, though, because we spend a lot of time in each node of the branch-and-bound tree to obtain it. As a consequence, our algorithm is outperformed in terms of CPU time by those reported on in the literature.

Run time profiles, branching decisions, and the average number of variables that is set by our pre-processing, reduced cost and logical implication subroutines are reported on in Table 4.5. The LP, F, VP, GP, Sep, Set, and Heur columns contain the percentage of CPU time spent in solving linear programs, formulating subproblems, initialising and maintaining variable pseudo costs, GUB pseudo costs, variable setting and our rounding heuristics, respectively. The GUB column contains the percentage of branches that were GUB branches, all other branches were variable branches. A minus indicates that all runs of the algorithm that were averaged over were solved in the root node. The U and L columns contain the average number of variables that were set by strengthened reduced cost fixing to their upper and lower bound, respectively. The SP, ND, and SD the average number of nodes that were set because they became simplicial, node-dominated, and set-dominated, respectively.

The GUB column clearly shows that GUB branches are competitive with variable branches when compared using pseudo costs. It was to be expected that GUB branching would be very useful on the dense instances. Surprisingly, it also is very competitive for the instances with $p = .1$ and $n \geq 90$. The relative large amount of time spent in initialising the GUB pseudo costs for problems with higher density ($p \geq .5$) is due to the fact that the branch-and-bound tree sizes are small on those classes of instances. Finally, we note that our variable setting criteria do not have a particularly large yield. On the other hand, they do not take up a lot of CPU time.

		Run-Time Profiles								Variable Setting				
n	p	LP	F	VP	GP	Sep	Set	Heur	GUB	U	L	SP	ND	SD
80	.1	0.3	3.7	40.5	12.6	36.5	1.9	1.7	7.8	0.0	2.2	0.2	0.4	0.0
	.2	1.8	6.2	15.9	24.4	37.9	4.2	3.8	22.2	0.0	0.0	0.0	0.0	0.0
	.3	3.4	8.8	15.0	25.3	31.1	5.6	4.4	18.5	0.0	0.0	0.0	0.0	0.0
	.4	2.8	10.3	15.8	29.2	27.4	5.6	3.4	24.8	0.0	0.0	0.0	0.0	0.0
	.5	1.9	10.0	18.3	37.2	21.9	4.1	2.5	25.1	0.0	0.0	0.0	0.0	0.0
	.6	1.2	7.9	20.3	44.4	18.8	2.9	1.7	19.2	0.0	0.0	0.0	0.0	0.0
	.7	0.6	5.8	23.6	47.3	17.7	2.3	1.0	21.2	0.0	0.0	0.0	0.0	0.6
	.8	0.2	4.3	21.0	52.0	17.8	2.3	0.6	31.8	0.0	0.1	0.0	0.0	8.8
	.9	0.1	9.1	0.0	0.0	56.4	12.2	1.2	–	0.0	15.2	0.0	2.5	24.7
90	.1	0.4	4.1	33.7	18.2	35.8	1.9	2.3	33.2	0.0	0.2	0.2	0.2	0.0
	.2	3.2	9.5	7.7	13.5	45.7	6.7	5.7	30.4	0.0	0.0	0.0	0.0	0.0
	.3	4.9	13.5	8.0	16.2	34.3	8.7	5.5	34.3	0.0	0.0	0.0	0.0	0.0
	.4	3.5	15.4	10.6	23.8	28.1	7.2	4.0	32.6	0.0	0.0	0.0	0.0	0.0
	.5	2.5	14.7	14.3	34.4	20.9	5.1	2.7	20.9	0.0	0.0	0.0	0.0	0.0
	.6	1.3	11.0	15.3	47.5	16.4	3.6	1.6	22.6	0.0	0.0	0.0	0.0	0.0
	.7	0.7	7.0	19.0	53.7	14.4	2.4	1.0	21.8	0.0	0.0	0.0	0.0	0.3
	.8	0.2	4.6	18.8	56.6	16.0	1.8	0.5	28.4	0.0	0.0	0.0	0.0	7.4
	.9	0.1	9.9	0.0	0.0	59.7	12.6	0.8	–	0.1	19.9	0.0	1.7	26.0
100	.1	0.7	6.5	13.1	13.0	52.2	3.1	2.9	71.5	0.0	0.0	0.1	0.2	0.0
	.2	5.9	12.9	4.3	9.6	39.9	8.9	8.7	25.0	0.0	0.0	0.0	0.0	0.0
	.3	7.4	20.1	4.2	12.0	27.8	10.4	6.7	21.4	0.0	0.0	0.0	0.0	0.0
	.4	6.2	24.3	5.7	16.8	22.3	9.6	3.9	16.7	0.0	0.0	0.0	0.0	0.0
	.5	3.1	18.9	11.5	32.6	18.3	6.3	2.6	21.9	0.0	0.0	0.0	0.0	0.0
	.6	1.7	14.2	14.1	43.4	16.1	4.7	1.5	21.7	0.0	0.0	0.0	0.0	0.0
	.7	0.7	9.0	16.9	53.2	14.6	2.6	0.9	21.8	0.0	0.0	0.0	0.0	0.0
	.8	0.1	3.5	17.0	67.6	9.6	1.2	0.4	46.8	0.0	0.0	0.0	0.0	4.7
	.9	0.0	4.3	10.5	54.4	22.4	4.0	0.3	100.0	0.0	15.0	0.0	0.6	26.1
110	.1	1.2	7.9	5.7	7.7	60.4	4.0	3.6	63.5	0.0	0.0	0.0	0.0	0.0
	.2	8.7	16.5	1.8	6.8	32.2	10.8	11.3	18.1	0.0	0.0	0.0	0.0	0.0
	.3	10.9	24.1	2.3	10.0	18.6	12.6	7.6	19.3	0.0	0.0	0.0	0.0	0.0
	.4	8.7	27.0	4.4	16.0	16.1	10.3	4.8	20.7	0.0	0.0	0.0	0.0	0.0
	.5	4.8	24.5	8.0	26.1	15.7	8.5	2.3	13.7	0.0	0.0	0.0	0.0	0.0
	.6	2.2	17.2	12.9	39.8	15.5	5.6	1.4	17.4	0.0	0.0	0.0	0.0	0.0
	.7	0.8	10.7	15.8	51.5	14.3	3.4	0.8	21.1	0.0	0.0	0.0	0.0	0.0
	.8	0.2	4.2	14.7	68.6	9.8	1.3	0.4	40.1	0.0	0.0	0.0	0.0	3.3
	.9	0.0	2.1	6.5	79.0	9.1	1.6	0.1	87.5	0.0	4.0	0.0	0.3	26.5

Table 4.5: Profiles, Branching, and Setting Statistics of Branch-and-Cut

Part II

Applications

Chapter 5

Map Labelling

5.1 Introduction

When designing maps, an important question is how to place the names of the cities on the map such that each name appears close to its city and no two names overlap. Various problems related to this question are referred to as *map labelling problems*. We consider a variant of the map labelling problem described in Section 5.1.1. In Section 5.2 we show that this variant of the map labelling problem reduces to a maximum independent set problem. To solve it to optimality we slightly refine our optimisation algorithms for finding maximum independent sets from Chapter 4 for map labelling. In Section 5.3 we present computational results of both our heuristics for finding independent sets from Chapter 4 and our refined optimisation algorithms. Our computational results are obtained on graphs derived from an interesting class of randomly generated map labelling problem instances that is taken from the literature. We show that our heuristic algorithms yield close to optimal solutions, and that our optimisation algorithms are capable of solving map labelling problem instances with up to 950 cities to optimality within reasonable time.

5.1.1 Problem Definition

The basic map labelling problem can be described as follows: given a set $P = \{\boldsymbol{p}^1, \boldsymbol{p}^2, \ldots, \boldsymbol{p}^n\}$ of n distinct points in $\mathbb{R}^2$, determine the supremum σ^* of all reals σ for which there exist n pairwise disjoint, axis-parallel $\sigma \times \sigma$ squares $Q_1, Q_2, \ldots, Q_n \subset \mathbb{R}^2$, where $\boldsymbol{p}^i$ is a corner of Q_i for all $i = 1, \ldots, n$. By "pairwise disjoint squares" we mean that no overlap between any two squares is allowed. Once the squares are known, they define the boundaries of the areas where the labels can be placed. We will refer to this problem as the *basic* map labelling problem. The *decision variant* of the map labelling problem is to decide, for any given σ, whether there exists a set of squares $Q_1, \ldots, Q_n$ as described above. Formann and Wagner [47] showed that this latter problem is $\mathcal{NP}$-complete.

Kučera, Mehlhorn, Preis, and Schwarzenecker [73] observed that there are only $O(n^2)$ possible values that σ^* can take. Optimising over those can be done by solving the decision variant only $O(\log n)$ times, using binary search with different values of σ. So, the basic map labelling problem reduces to the decision variant. Kučera *et al.* also present an exponential-time algorithm that solves the decision variant of the map labelling problem to optimality. They do not, however, report on successful computational experiments.

Another variant of the map labelling problem, which we will refer to as the *optimisation variant*, has the label size σ as input, and asks for as many pairwise disjoint squares of the desired characteristic as possible. If the number of squares in an optimal solution to the optimisation variant of the map labelling problem equals n, then this solution is a feasible solution to the decision variant, and vice versa. Hence, the decision variant of the map labelling problem reduces to the optimisation variant via a polynomial reduction. It is the optimisation variant of the map labelling problem that is the subject of this chapter. Since the decision variant is $\mathcal{NP}$-complete, the optimisation variant is $\mathcal{NP}$-hard.

5.1.2 Related Literature

A recent survey on map labelling is given by Neyer [89]. An exhaustive bibliography concerning map labelling is maintained by Wolff and Strijk [121]. Formann and Wagner [47] developed a $\frac{1}{2}$-approximation algorithm for the optimisation variant of the map labelling problem. Different heuristic algorithms (including simulated annealing) are discussed by Christensen, Marks, and Shieber [27]. Van Dijk, Thierens and de Berg [41] considered genetic algorithms, Wagner and Wolff [119] propose a hybrid heuristic. Cromly [30] proposed a semi-automatic LP-based approach for finding feasible solutions to the optimisation variant. Zoraster [127, 128] used Lagrangian relaxation to make a heuristic algorithm for the optimisation variant.

All the results mentioned so far concern the problem in which each point is labelled with one (square) region from a finite set of candidate (square) regions. Kakoulis and Tollis [68] exploit this discrete nature to unify several slightly more general map labelling problems. A different approach reported on by van Kreveld, Strijk, and Wolff [72] and Klau and Mutzel [71] is to allow a label to take any position, as long as its corresponding point is on its boundary. This leads to so-called *sliding* map labelling models. The advantage of a sliding model is that more labels can be placed without overlap. In more recent studies also other shapes of the label regions are considered (see e.g. Qin, Wolff, Xu and Zu [97]).

In this chapter we study the optimisation variant of the basic map labelling problem. In the remainder of this chapter we will refer to it as the map labelling problem.

5.2 A Branch-and-Cut Algorithm

In this section, we present our branch-and-cut algorithm for map labelling. Section 5.2.1 gives the reduction from the map labelling problem to the maximum independent set problem. The reduction uses the notion of *conflict graphs* of map labelling problem instances. In Section 5.2.2 we show that there are at most polynomially many maximal cliques in the conflict graph of a map labelling problem instance, and that these can be found in polynomial time.

Our algorithms for the map labelling problem work with the conflict graph only, and are therefore applicable to all map labelling problems that can be formulated as maximum independent set problems on conflict graphs. The problems that fit in the framework by Kakoulis and Tollis [68] all satisfy this requirement. For example, map labelling problems with finite, but more than four possible placements of each label, and problems in which the regions occupied by labels have different (non-uniform) shapes and sizes can all be formulated as a maximum independent set problems on conflict graphs. Also, it is possible to assign different weights to different label placements, in which case we would be interested in a maximum weight independent set. We stress that all our algorithms for map labelling can easily be transferred to these more general models. To simplify the discussion and to allow for comparison of our results with results reported on in the literature we focus on the special case in which there are four possible positions for each label, and the labels are rectangles of given dimensions.

The conflict graphs arising from map labelling problems have a very special characteristic, namely, it is easy to embed them in the plane. In the resulting embedded graphs, nodes are only connected to other nodes that are nearby. It is this local structure in which the independent set problems arising from map labelling differ from those for the uniform random graphs studied in Chapter 4. We exploit the special structure in two ways.

It turns out that our routines for setting variables (see Sections 3.2.3, 4.3.1 and 4.3.5) are effective for independent set problems in conflict graphs of map labelling problem instances. In combination with the local structure this creates the possibility that the sub-graph induced in the conflict graph by the nodes corresponding to free variables can become disconnected in any iteration of the branch-and-cut algorithm, even when we start with a connected conflict graph. In those iterations the associated maximum independent set problems can be decomposed. We exploit this observation and enhance our branch-and-cut algorithm with a technique that we call *variable setting by recursion and substitution*, which is the subject of Section 5.2.3.

A second way to exploit the local structure in the conflict graphs of map labelling problems is to be more conservative with calling the separation routines. Especially for large problems this can be a key factor in the effectiveness of a branch-and-cut algorithm. Section 5.2.4 explains how we enhance our branch-and-cut algorithm with this idea.

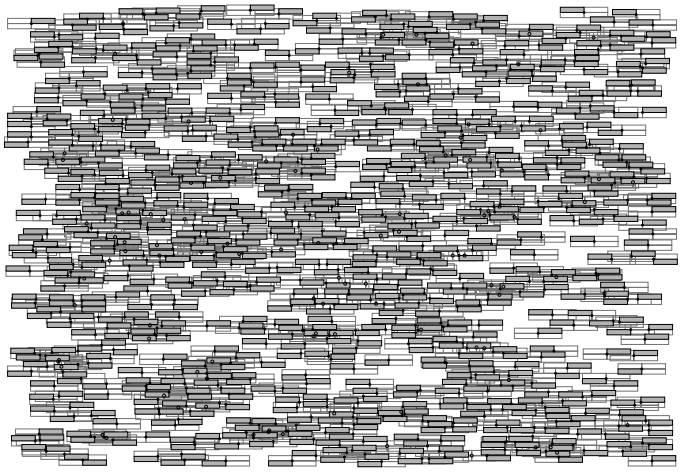


Figure 5.1: A Solution to a Map Labelling Problem on 950 Cities

5.2.1 Reduction from Map Labelling to Independent Set

An instance of the map labelling problem in which the labels are rectangles of uniform size consists of a finite set of points $P \subset \mathbb{R}^2$ and a label size $\sigma_1 \times \sigma_2$. In the following, we assume that the label size (σ_1, σ_2) is fixed. Under this assumption, an instance of the map labelling problem is completely specified by the point set P.

For any point $\boldsymbol{p} \in P$, there are four possible placements of a rectangular label Q such that $\boldsymbol{p}$ is a corner of Q, each occupying a different rectangular region in $\mathbb{R}^2$. Denote these rectangular regions by $Q_{\boldsymbol{p}i}$ with $i \in \{1, 2, 3, 4\}$. Let $\mathcal{Q}(P)$ be the set of all rectangular regions that correspond to possible label placements in a solution to the map labelling problem instance P:

$$\mathcal{Q}(P) = \{Q_{\boldsymbol{p}i} \mid \boldsymbol{p} \in P,\, i \in \{1, 2, 3, 4\}\}.$$

A *feasible solution* to the map labelling problem instance P is a set of rectangular regions $\mathcal{S} \subseteq \mathcal{Q}(P)$ such that for any pair of rectangular regions $Q, R \in \mathcal{S}$ we have that if $Q \cap R \neq \varnothing$ then $Q = R$. Note that this definition correctly prevents a point $\boldsymbol{p} \in P$ from having more than one label, as the intersection of those labels would contain $\boldsymbol{p}$ and would therefore be non-empty. The map labelling problem is the problem of finding a solution of maximum cardinality. An optimal solution to a map labelling problem on 950 cities is depicted in Figure 5.1. All labelled cities are drawn as black discs, all unlabelled cities are drawn as black circles. All possible rectangular regions representing a label position are shown, those that are coloured grey are part of the solution.

To reduce the map labelling problem to a maximum independent set problem, we will make use of the following notion from graph theory. In the definition we will take the more general perspective of sets of regions. Also, v_Q always denotes some node uniquely associated with the region Q.

Definition 5.1. Given a set of regions $\mathcal{Q}$, the *intersection graph* $G_{\mathcal{Q}} = (V_{\mathcal{Q}}, E_{\mathcal{Q}})$ is given by

$$V_{\mathcal{Q}} = \{v_Q \mid Q \in \mathcal{Q}\}, \text{ and} \tag{5.1}$$
$$E_{\mathcal{Q}} = \{\{v_Q, v_R\} \subseteq V_{\mathcal{Q}} \mid Q \cap R \neq \varnothing\}. \tag{5.2}$$

Here, (5.1) ensures that for each region in $\mathcal{Q}$ there is a node v_Q in $G_{\mathcal{Q}}$, and (5.2) ensures that for each pair (Q, R) of intersecting regions in $\mathcal{Q}$ there is an edge $\{v_Q, v_R\}$ in $G_{\mathcal{Q}}$. Given an instance P of the map labelling problem, we define the *conflict graph* of P as the intersection graph of $\mathcal{Q}(P)$.

Theorem 5.1. *Let $P \subset \mathbb{R}^2$ be an instance of the map labelling problem. There is a bijection between the feasible solutions $\mathcal{S}$ of P and the independent sets in the conflict graph of P.*

Proof. Suppose that $\mathcal{S}$ is a solution of P. Let $S \subseteq V_{\mathcal{Q}(P)}$ be the unique set of nodes of the conflict graph of P that correspond to the labels in $\mathcal{S}$, i.e. $S = \{v_Q \mid Q \in \mathcal{S}\}$. Because $\mathcal{S}$ is a solution to P, for any two distinct labels $Q, R \in \mathcal{S}$ we have that $Q \cap R = \varnothing$. Hence $\{v_Q, v_R\} \notin E_{\mathcal{Q}}$. It follows that S is an independent set in $G_{\mathcal{Q}(P)}$.

Now, suppose that S is an independent set in $G_{\mathcal{Q}(P)}$. Let $\mathcal{S}$ be the unique set of rectangular regions associated with the nodes in S, i.e. $\mathcal{S} = \{Q \mid v_Q \in S\}$. Because S is an independent set in $G_{\mathcal{Q}}$ and $\{v_Q, v_R\} \in E_{\mathcal{Q}}$ for all distinct $Q, R \in \mathcal{Q}$ such that $Q \cap R \neq \varnothing$ we have that $Q \cap R = \varnothing$ for all distinct $Q, R \in \mathcal{S}$. Hence $\mathcal{S}$ is a solution to P. □

An example of a conflict graph of a map labelling instance is depicted in Figure 5.2. The map labelling instance has 250 cities, the conflict graph has 1000 nodes. A maximum independent set in the conflict graph is indicated by the solid nodes.

5.2.2 Finding all Maximum Cliques in the Conflict Graph

Suppose we are given a set $\mathcal{Q}$ of axis-parallel rectangular regions in $\mathbb{R}^2$. It was observed by Imai and Asano [63] that there exists a correspondence between the cliques in $G_{\mathcal{Q}}$ and maximal non-empty regions that are contained in the intersection of subsets of regions from $\mathcal{Q}$.

Lemma 5.2. *Let $\mathcal{Q}$ be a set of axis-parallel rectangular regions. Then, there is a bijection between the cliques in $G_{\mathcal{Q}}$ and the non-empty intersections of subsets of rectangular regions from $\mathcal{Q}$.*

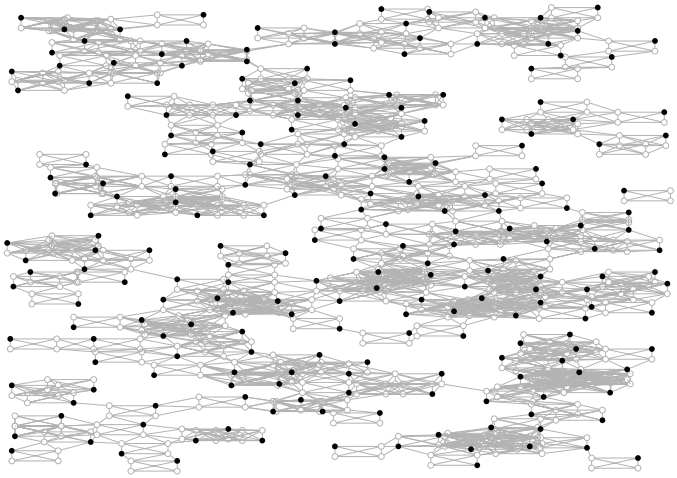


Figure 5.2: The Conflict Graph of a Map Labelling Problem on 250 Cities

Proof. Let $\mathcal{S} \subseteq \mathcal{Q}$ be a collection of rectangular regions with non-empty intersection. Clearly, the rectangular regions in $\mathcal{S}$ intersect pairwise. Hence, for each $Q, R \in \mathcal{S}$, $\{v_Q, v_R\} \in E_{\mathcal{Q}}$. So, the set $C = \{v_Q \mid Q \in \mathcal{S}\}$ is a clique in $G_{\mathcal{Q}}$.

Conversely, let C be a clique in $G_{\mathcal{Q}}$, and let $\mathcal{S} = \{Q \mid v_Q \in C\}$ be the rectangular regions in $\mathcal{Q}$ that correspond to the nodes in C. Because C is a clique, all rectangular regions in $\mathcal{S}$ intersect pairwise. This implies that the region $R = \bigcap_{Q \in \mathcal{S}} Q$ is non-empty by the following argument (see e.g. Danzer and Grünbaum [37]). The intersection of a set of axis-parallel rectangular regions is non-empty if and only if the intersection of the projections of the rectangular regions on the axis are non-empty. The projections of the rectangular regions on the axes are sets of line-segments in $\mathbb{R}$. By Helly's theorem, a set of line-segments has a non-empty intersection if and only if all pairs of line-segments in the set intersect. □

Let $\mathcal{C}$ be the collection of all maximal cliques in $G_{\mathcal{Q}}$, and let

$$\mathcal{R} = \{\textstyle\bigcap_{v_Q \in C} Q \mid C \in \mathcal{C}\}.$$

It follows from the proof of Lemma 5.2 that $\mathcal{R}$ contains all maximal regions that are contained in the intersection of a maximal subset of rectangular regions from $\mathcal{Q}$ that has a non-empty intersection. The following lemma shows that the regions in $\mathcal{R}$ are mutually disjoint.

Lemma 5.3. *Let $R_1, R_2 \in \mathcal{R}$. If $R_1 \cap R_2 \neq \varnothing$, then $R_1 = R_2$.*

Proof. Let $\mathcal{S}_i \subseteq \mathcal{Q}$ be the set of rectangular regions such that $R_i = \bigcap_{Q \in \mathcal{S}_i} Q$ ($i \in \{1, 2\}$), and assume that $R_1 \cap R_2 \neq \varnothing$. Let $C = \{v_Q \mid Q \in \mathcal{S}_2\}$. Note that C is a maximal clique in $G_\mathcal{Q}$ because $R_2 \in \mathcal{R}$. Choose $Q_1 \in S_1$ arbitrarily. Because $R_1 \cap R_2 \neq \varnothing$, also $Q_1 \cap R_2 \neq \varnothing$, which in turn implies that $Q_1 \cap Q \neq \varnothing$ for all $Q \in \mathcal{S}_2$. This implies that $\{v_{Q_1}, v_Q\} \in E_\mathcal{Q}$ for all $Q \in \mathcal{S}_2 \setminus \{Q_1\}$. Because C is a maximal clique, this means that $v_{Q_1} \in C$, so $Q_1 \in \mathcal{S}_2$. Since Q_1 was chosen arbitrarily, it follows that $\mathcal{S}_1 \subseteq \mathcal{S}_2$. Similarly, we find that $\mathcal{S}_2 \subseteq \mathcal{S}_1$. This implies that $\mathcal{S}_1 = \mathcal{S}_2$ and that $R_1 = R_2$. □

Recall the clique formulation of the independent set problem on G from Section 4.1.2. If the clique formulation is derived from the collection of all maximal cliques in G we call it the *complete clique formulation*. We are interested in finding all maximal cliques in the intersection graph of $\mathcal{Q}$ in order to obtain the complete clique formulation of the independent set problem on $G_\mathcal{Q}$. The following lemma states that the number of maximal cliques in the intersection graph of $\mathcal{Q}$ is not too large.

Lemma 5.4. *The number of maximal cliques in $G_\mathcal{Q} = (V_\mathcal{Q}, E_\mathcal{Q})$ is at most $|V_\mathcal{Q}| + |E_\mathcal{Q}|$.*

Proof. We will show that there exists an injection $f : \mathcal{R} \to V_\mathcal{Q} \cup E_\mathcal{Q}$. Since $|\mathcal{R}| = |\mathcal{C}|$, this proves the lemma.

Consider any region $R \in \mathcal{R}$. Since R is the intersection of a subset of rectangular regions from $\mathcal{Q}$ we can write $R = \{\boldsymbol{x} \in \mathbb{R}^2 \mid \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\}$, where l_1 and l_2 are determined by the boundary of some rectangular regions $Q_1, Q_2 \in \mathcal{Q}$, respectively. Note that Q_1 and Q_2 do not have to be unique. Consider any pair Q_1, Q_2 as above. If $Q_1 = Q_2$, we define $f(R)$ to be v_{Q_1}. Otherwise, $\boldsymbol{l} \in Q_1 \cap Q_2$, so $\{v_{Q_1}, v_{Q_2}\} \in E_\mathcal{Q}$, and we define $f(R)$ to be $\{v_{Q_1}, v_{Q_2}\}$.

It remains to show that f is indeed an injection. Let $R_1, R_2 \in \mathcal{R}$ and suppose that $f(R_1) = f(R_2)$. Let $\boldsymbol{l}^i, \boldsymbol{u}^i \in \mathbb{R}^2$ be the points such that $R_i = \{\boldsymbol{x} \in \mathbb{R}^2 \mid \boldsymbol{l}^i \leq \boldsymbol{x} \leq \boldsymbol{u}^i\}$ (for $i \in \{1, 2\}$). Because $f(R_1) = f(R_2)$, by construction of f we have that $\boldsymbol{l}^1 = \boldsymbol{l}^2$, so $R_1 \cap R_2 \neq \varnothing$. By Lemma 5.3 we find that $R_1 = R_2$. Hence f is an injection. □

All single-node components of $G_\mathcal{Q}$ define their own maximal clique. As these maximal cliques can be reported separately, suppose that $G_\mathcal{Q}$ does not have isolated nodes. The following algorithm is a brute-force algorithm by Strijk [108] that exploits Lemma 5.3 to find all maximal cliques. For each edge $\{v_Q, v_R\} \in E_\mathcal{Q}$, we compute a maximal clique C that contains a fixed corner $\boldsymbol{l}$ of $Q \cap R$, and then test whether C is a maximal clique in $G_\mathcal{Q}$. If this is the case, we report C, otherwise we proceed with the next edge. Computing C can be done by testing for each node $v_S \in N(\{v_Q, v_R\})$ whether $\boldsymbol{l} \in S$ and if so, adding v_S to C. C is a maximal clique in $G_\mathcal{Q}$ if for each node $v_S \in N(\{v_Q, v_R\})$ we have that $S \cap \bigcap_{v_Q \in C} Q = \varnothing$.

Theorem 5.5. *The maximal cliques in $G_\mathcal{Q}$ can be found in $O(|V_\mathcal{Q}||E_\mathcal{Q}|)$ time.*

Proof. The correctness of the brute-force algorithm described above follows directly from Lemma 5.2 and Lemma 5.3. It remains to analyse its time complexity. For each edge $\{u, v\} \in E_{\mathcal{Q}}$, the brute-force algorithm described above uses $O(|\delta(u)| + |\delta(v)|)$ time. Hence, the total time complexity needed for reporting all maximal cliques in $G_{\mathcal{Q}}$ is

$$\sum_{\{u,v\}\in E_{\mathcal{Q}}} O(|\delta(u)| + |\delta(v)|) = O(\sum_{\{u,v\}\in E_{\mathcal{Q}}} |\delta(u)| + |\delta(v)|) = O(\sum_{u\in V_{\mathcal{Q}}} \sum_{e\in\delta(u)} |\delta(u)|)$$
$$= O(\sum_{u\in V_{\mathcal{Q}}} |\delta(u)|^2) \leq O(|V_{\mathcal{Q}}||E_{\mathcal{Q}}|).$$

□

The direct consequence of Theorem 5.5 is that, given an instance of the map labelling problem P, we can compute the complete clique formulation of the independent set problem in $G_{\mathcal{Q}(P)}$ in $O(|V_{\mathcal{Q}(P)}||E_{\mathcal{Q}(P)}|)$ time.

5.2.3 Variable Setting by Recursion and Substitution

Suppose we are solving an instance of the map labelling problem P with conflict graph $G = (V, E)$ using our branch-and-cut algorithm from Section 4.3 on G. Focus on iteration j of the branch-and-cut algorithm just after the termination of the cutting plane algorithm. Recall from Section 4.3.5 that we have solved an LP of the form (4.15) for some lower and upper bounds $\boldsymbol{l}, \boldsymbol{u}$. Let $J_1, J_2 \subseteq V$ be the set of nodes whose corresponding variables are set to zero, and one, respectively, and let $F = V \setminus (J_1 \cup J_2)$ be the set of nodes whose corresponding variables are free. Because we exhaustively apply Proposition 4.13, for each $v \in J_2$ we have that $\boldsymbol{l}_{N(v)} = \boldsymbol{0}$, so all neighbours of v are members of J_1. Note that $G[F]$ does not need to be a connected graph, even if G is. Let $\{F_1, \ldots, F_k\}$ be the partition of F such that for each $i \in \{1, \ldots, k\}$, $G[F_i]$ is a connected component of $G[F]$. The following proposition, that is similar to Proposition 4.7, states that we can find the optimal integer solution in the set

$$P_E \cap \{\boldsymbol{x} \in \mathbb{R}^V \mid \boldsymbol{l}_V \leq \boldsymbol{x} \leq \boldsymbol{u}_V\}$$

by combining optimal solutions for the individual F_i, where P_E again denotes the edge-formulation of Section 4.1.2.

Proposition 5.6. *Let $i \in \{1, \ldots, k\}$, let J_1, J_2 and F_i be as above, and let $I \subseteq F_i$ be a maximum independent set in $G[F_i]$. Then, there exists an independent set I^* in G with $I \subseteq I^*$ that is maximum under the restrictions that $J_1 \cap I^* = \varnothing$ and $J_2 \subseteq I^*$.*

Proof. Let I_1 be a maximum independent set in G under the restrictions that $J_1 \cap I_1 = \varnothing$ and $J_2 \subseteq I_1$. Let $I_2 = I_1 \setminus F_i$ and $I^* = I_2 \cup I$. Because we exhaustively use Proposition 4.13 and F_i is maximal by the definition of connected components, for all $v \in F_i$ and $\{v, w\} \in \delta(v)$ we have that $w \in F_i \cup J_1$. It

follows that I^* is an independent set in G. Because $I_1 \cap F_i$ is an independent set in $G[F_i]$, we find that

$$|I_2| = |I_1 \setminus F_i| = |I_1| - |I_1 \cap F_i| \geq |I_1| - |I|.$$

Because $I \cap I_2 = \varnothing$, this implies that

$$|I^*| = |I_2 \cup I| = |I_2| + |I| \geq |I_1| - |I| + |I| \geq |I_1|.$$

Moreover, $J_1 \cap I^* = \varnothing$ and $J_2 \subseteq I^*$ by choice of I_1 and construction of I^*. So I^* is an independent set in G that is at least as large as I_1, and satisfies $I \subseteq I^*$, $J_1 \cap I^* = \varnothing$ and $J_2 \subseteq I^*$. □

Now, suppose that $|F_1| \geq |F_i|$ for all $i \in \{2, \ldots, k\}$. We see F_1 as the main component of the problem we solve in the part of the branch-and-bound tree rooted at v_j, and leave it aside for the remainder of iteration j. (Here v_j again denotes the node of the branch-and-bound tree corresponding to iteration j.) For all $i \in \{2, \ldots, k\}$, however, we recursively compute a maximum independent set I_i^* in $G[F_i]$, and then set the variables in I_i^* to one and the variables in $F_i \setminus I_i^*$ to zero. By Proposition 5.6, this maintains condition (2.3) as an invariant. After substituting the solutions of the recursively computed independent sets, we obtain a new solution and a new bound for iteration j of the branch-and-cut algorithm. We call this technique *variable setting by recursion and substitution*, or SRS.

5.2.4 Separation for Map Labelling Problems

We can exploit the structure of the independent set problems arising from map labelling by more conservatively calling our separation routines. Because adding cuts or bounds only tends to change the LP solution in a small region surrounding the nodes that are directly effected by them, there is no need to call the separation routines again in parts of the graph where no change of the LP solution has occurred.

We say that a solution $\boldsymbol{x}$ *precedes* $\tilde{\boldsymbol{x}}$ in a branch-and-cut algorithm if $\boldsymbol{x}$ ($\tilde{\boldsymbol{x}}$) is the optimal solution to an LP relaxation P ($\tilde{P}$, respectively), and $\tilde{P}$ is derived in the branch-and-cut algorithm from P by adding cuts or variable bounds. Focus on iteration i of the branch-and-cut algorithm, and let v_j be the parent of v_i in the branch-and-bound tree if $i > 1$. If the current LP relaxation is $\tilde{P}$ and $\tilde{P}$ is not the first iteration of the cutting plane algorithm in iteration i of the branch-and-cut algorithm, then P is the LP relaxation associated with the previous iteration of the cutting plane algorithm. Otherwise, if $i > 1$ and the current iteration of the cutting plane algorithm is its first iteration in iteration i of the branch-and-cut algorithm, then P is the LP relaxation associated with the last iteration of the cutting plane algorithm in iteration j of the branch-and-cut algorithm.

Let $G = (V, E)$ denote the conflict graph of an instance of the map labelling problem. Recall the definition of P_E from Section 4.1.2. Focus on iteration i of

the branch-and-cut algorithm. Suppose that $\boldsymbol{x} \in P_E$ is an optimal solution to an LP relaxation in iteration i of the branch-and-cut algorithm, and let $\boldsymbol{x}' \in P_E$ be the optimal LP solution that precedes $\boldsymbol{x}$. In each iteration of the branch-and-cut algorithm, we only call the separation algorithm for lifted odd hole inequalities presented in Section 4.3.3 starting from nodes $v \in V$ for which x_v is fractional and $x_v \neq x'_v$. In this way, we avoid doing the same calculations over and over again in each iteration of the branch-and-cut algorithm. Although there is no guarantee that we find all the lifted odd hole inequalities one could find if one would start the separation from all nodes that correspond to fractional variables, we believe that our cutting plane algorithm still finds most of them. The decrease in processing time needed for each iteration of our branch-and-cut algorithm is considerable.

We now focus our attention on the remaining separation algorithms. For large map labelling instances, the systems of congruences mod-k that we have to solve for each run of the mod-k separation algorithm from Section 3.3.5 are large as well. As a consequence, the mod-k separation turns out to be very time consuming. Although we suspect that a similar incremental strategy as for separating lifted odd-holes would solve this problem, developing such a strategy is non-trivial, and is still on our to-do list. Therefore, in our implementation we restrict the separation of mod-k inequalities to iteration one of the branch-and-cut algorithm. Since we start out with the complete clique formulation, we do not call our separation routines for finding maximal clique inequalities.

5.3 Computational Results

We tested our heuristic and optimisation algorithms on the same class of map labelling instances as used by Christensen *et al.* [27] and van Dijk *et al.* [41]. These instances are generated by placing n (integer) points on a standard map of size 792 by 612. The points have to be labelled using labels of size 30×7[1]. For each $n \in \{100, 150, \ldots, 750, 950\}$ we randomly generated 25 maps. We will first turn our attention to the computational behaviour of our heuristic algorithms, then we will discuss our optimisation algorithms, and finally we will discuss the influence of SRS.

Heuristics

We evaluated the performance of our heuristics for the maximum independent set problem on conflict graphs of map labelling problem instances. The algorithms are summarised in Table 5.1 (a copy of Table 4.1). All LP-based rounding

[1]Since we use closed labels, the label $Q_1 = \{\boldsymbol{x} \in \mathbb{R}^2 \mid \boldsymbol{0} \leq \boldsymbol{x} \leq (30, 7)^T\}$ intersects the label $Q(\boldsymbol{\alpha}) = \{\boldsymbol{x} \in \mathbb{R}^2 \mid \boldsymbol{\alpha} \leq \boldsymbol{x} \leq (\alpha_1 + 30, \alpha_2 + 7)^T\}$ for all $\boldsymbol{\alpha}$ with $|\boldsymbol{\alpha}| \leq (30, 7)^T$. From the integrality of the data it follows that either $|\alpha_1| \geq 31$, or $|\alpha_2| \geq 8$, or both if Q_1 and $Q(\boldsymbol{\alpha})$ do not intersect. Some researchers see this as a reason to actually use labels of size $(30-\epsilon, 7-\epsilon)^T$ for some small constant $\epsilon > 0$. The reader should be aware of this when comparing the results presented in this chapter to results found in the literature.

Name	Heuristic	Section
O1	1-Opt starting from zero	4.2.3
O2	1-Opt followed by 2-opt starting from zero	4.2.3
L1	Simple LP rounding followed by 1-opt	4.2.1, 4.2.3
L2	Simple LP rounding followed by 1-opt and 2-opt	4.2.1, 4.2.3
R1	Minimum regret rounding with parameter $t = 1$	4.2.2
R2	Minimum regret rounding with parameter $t = 2$	4.2.2

Table 5.1: Independent Set Heuristics Applied to Map Labelling Instances

algorithms are started from a fractional independent set computed by the cutting plane algorithm as described in Section 5.2.4, using the complete clique formulation.

The average number of labels placed by our heuristics and the corresponding running times are reported in Table 5.2. The columns O1 and O2 refer to the iterative improvement algorithms that were presented in Section 4.2.3. These algorithms use the 1-opt and 2-opt neighbourhoods, respectively, starting from $\mathbf{0}$. The L1, L2, R1 and R2 columns all refer to LP-based rounding algorithms. The L1 and L2 columns refer to the algorithms that first apply the simple LP rounding algorithms from Section 4.2.1 to obtain an integer solution. Next, they invoke an iterative improvement algorithm starting from this integer solution using the 1-opt and 2-opt neighbourhoods, respectively. The R1 and R2 columns refer to algorithms that apply the minimum regret rounding heuristic from Section 4.2.2 to the optimal solution of the LP relaxation, with the parameter t equal to 1 and 2, respectively. The α columns contain the average number of labels placed, the CPU columns contain the average number of CPU seconds that were needed to find the reported independent sets, the time needed to solve the LP relaxations excluded. Finally, the $\alpha(G)$ column contains the average optimal number of labels that can be placed in the test instances.

It is clear from Table 5.2 that all rounding heuristics are very fast in practice. Their running time is negligible for instances with up to 950 cities. Again, in terms of solution quality 2-opt outperforms 1-opt. Using an LP solution as a starting point helps in improving the solution quality, and minimum regret rounding outperforms the simple rounding heuristics. The minimum regret heuristics with parameters $t = 2$ and $t = 1$ both perform equally well. The average number of labels placed by the minimum regret heuristics is within four tenth of the average number of labels in the optimal solutions for all choices of n, and strictly within one tenth for all choices of $n \leq 850$, with the parameter $t = 2$.

Before one can apply the rounding heuristics, one has to solve an LP relaxation. To be completely honest, we stress that solving the strengthened LP relaxation that we use is time consuming (just take a look at Figure 5.3), and would dominate the running time of a stand-alone heuristic that first solves a strengthened LP relaxation and then performs the rounding, for example using

	O1		O2		L1		L2		R1		R2		
n	α	CPU	α	CPU	α	CPU	α	CPU	α	CPU	α	CPU	$\alpha(G)$
100	97.5	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0
150	144.2	0.0	149.5	0.0	149.9	0.0	149.9	0.0	149.9	0.0	149.9	0.0	149.9
200	189.0	0.0	198.8	0.0	199.9	0.0	199.9	0.0	199.9	0.0	199.9	0.0	199.9
250	232.8	0.0	247.6	0.0	249.6	0.0	249.6	0.0	249.6	0.0	249.6	0.0	249.6
300	274.9	0.0	294.3	0.0	299.2	0.0	299.2	0.0	299.2	0.0	299.2	0.0	299.2
350	314.1	0.0	340.1	0.0	348.5	0.0	348.5	0.0	348.5	0.0	348.5	0.0	348.5
400	354.2	0.0	386.0	0.0	397.7	0.0	397.7	0.0	397.7	0.0	397.7	0.0	397.7
450	388.7	0.0	428.4	0.0	445.1	0.0	445.1	0.0	445.1	0.0	445.1	0.0	445.1
500	422.8	0.0	469.8	0.0	492.9	0.0	492.9	0.0	492.9	0.0	492.9	0.0	492.9
550	456.9	0.0	510.1	0.0	539.6	0.0	539.7	0.0	539.7	0.0	539.7	0.0	539.7
600	486.0	0.0	546.6	0.0	582.9	0.0	582.9	0.0	582.9	0.0	582.9	0.0	582.9
650	518.4	0.0	584.6	0.1	627.2	0.0	627.2	0.0	627.3	0.0	627.3	0.0	627.3
700	545.3	0.0	619.4	0.1	670.2	0.0	670.2	0.0	670.4	0.0	670.4	0.0	670.4
750	571.8	0.0	651.6	0.1	708.8	0.0	709.0	0.0	709.2	0.0	709.2	0.0	709.2
800	600.9	0.0	684.1	0.1	748.1	0.0	748.9	0.0	749.2	0.0	749.3	0.0	749.3
850	622.9	0.0	712.9	0.1	784.7	0.0	785.6	0.0	785.9	0.0	786.0	0.0	786.0
900	639.9	0.0	738.5	0.1	815.6	0.0	817.4	0.0	818.1	0.0	818.1	0.0	818.2
950	664.7	0.0	762.7	0.1	849.0	0.0	851.0	0.0	852.3	0.0	852.2	0.0	852.6

Table 5.2: Results of Independent Set Heuristics on Map Labelling Instances

the minimum regret rounding algorithm. On the other hand, the solutions that can be obtained this way are worth waiting for.

Optimisation Algorithms

As in Chapter 4, we tested our cutting plane/branch-and-bound algorithm and our branch-and-cut algorithm for the map labelling problem on the same set of instances that we used to evaluate the performance of our heuristics. The average running times and branch-and-bound tree sizes of these algorithms can be found in Figure 5.3. For the branch-and-cut algorithm, problem statistics and the gaps closed by the various valid inequalities are depicted in Figure 5.4, run time profiling data is given in Figure 5.5, the relative number of variable and GUB branches and the average number of the mod-k cuts found for each value of k are given in Figure 5.6, and finally the average number of variables set by pre-processing, reduced cost, and logical implication sub-routines are depicted in Figure 5.7.

The first thing to note from Figure 5.3 is that the branch-and-bound tree does not start to grow until the number of cities to label is approximately 750, and then it starts to grow at a tremendous rate. There is an obvious explanation for this, namely, that we do increase the number of cities that we label but we do not increase the map, so the problems become more dense, and 750 cities seems to be a critical number. Taking this into account, the steady exponential growth of the running time needed seems to be remarkable. The exponential

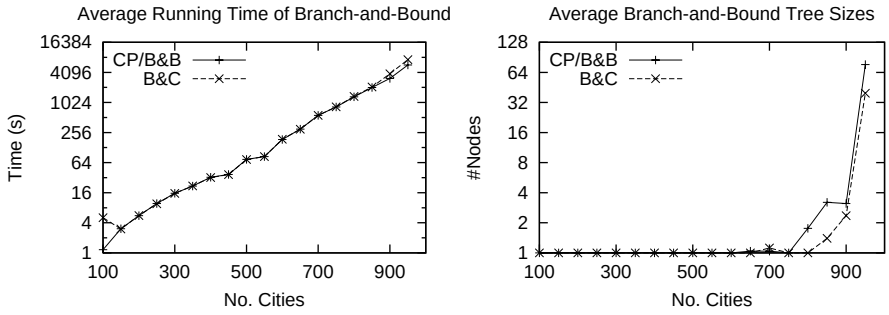


Figure 5.3: Average Branch-and-Bound Running Time and Tree Sizes

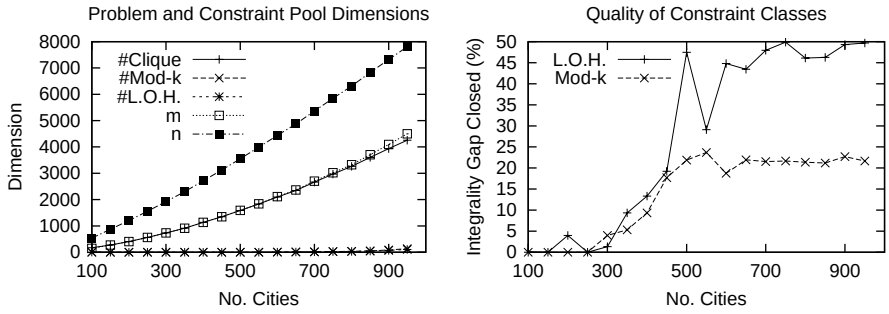


Figure 5.4: Problem Statistics and Constraint Performance

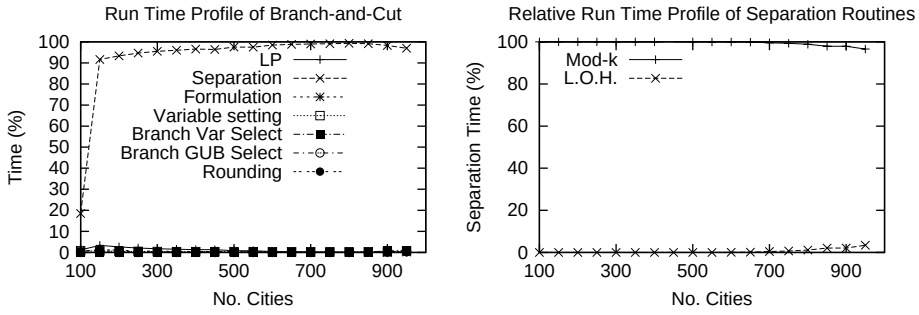


Figure 5.5: Run Time Profiling Data

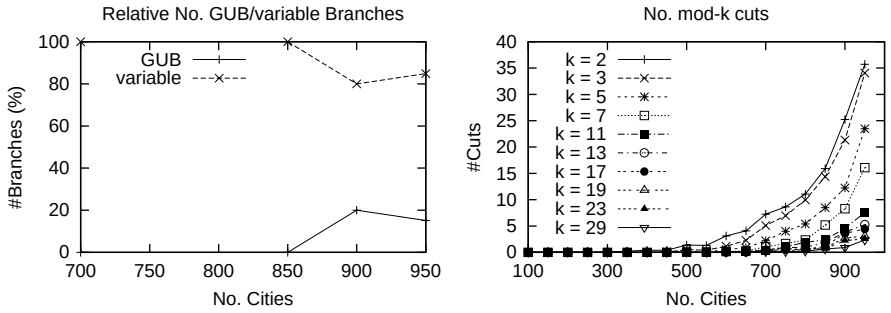


Figure 5.6: Branching Decisions and Mod-k Statistics

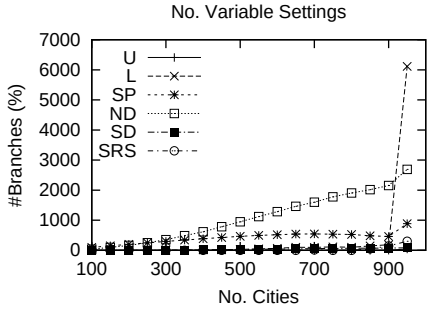


Figure 5.7: Number of Variable Setting Decisions

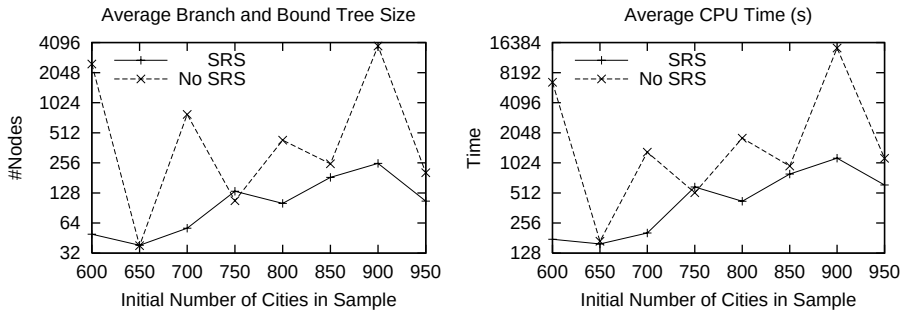


Figure 5.8: Influence of SRS on the Behaviour of the Branch-and-Cut Algorithm

growth of the running time demonstrates that the cutting plane algorithm itself behaves exponentially.

For the purpose of testing the influence of SRS we did some experiments in which we scaled the size of the map in order to keep the density of the graphs constant. We discuss these experiments later in this section in detail. What is important here is to note that they also show that the tremendous growth does not occur if the density is kept constant; instead we see a more modest exponential growth (see Figure 5.8).

From Figure 5.4 it is clear that although the number of lifted odd hole inequalities is small, their contribution to the quality of the LP formulation is significant. The same holds for mod-k cuts, although to a lesser extent. From Figure 5.5 it is clear that most of the running time is spent in the separation of mod-k cuts. We feel that this is due to the fact that we do not exploit the similarity between consecutive LP solutions in the branch-and-cut algorithm as we do with the lifted odd hole separation as discussed in Section 5.2.4.

We end the discussion of our optimisation algorithm by considering Figures 5.6 and 5.7. GUB branching does contribute a little, but most of the time it was more attractive to branch on a variable when comparing branches on pseudo-cost. We believe that this is due to the relative small size of the cliques in conflict graphs of map labelling instances. Figure 5.7 shows the average number of times that a variable can be set. Here U, L, SP, D, SD, and SRS indicate setting by reduced cost to upper bound, to lower bound, setting of simplicial nodes, dominated nodes, set dominated nodes and SRS, respectively. The spike in the graph of setting variables to their lower bound is caused by a single run in which this could be applied frequently in the lower parts of the tree. SRS only starts to play a role for the larger instances.

SRS

To evaluate the influence that SRS can have on the performance of our algorithm, we conducted a second set of experiments. These experiments were conducted with an earlier version of our code that did not feature the strengthened reduced cost setting from Section 3.2.3, the pre-processing from Section 4.3.1, the complete GUB formulation of Section 5.2.2 and the separation of mod-k inequalities. In these experiments the density of the problems was kept constant by making the map size a function of the number of cities. For a problem with n cities, we use a map of size $\lfloor 792/\sqrt{750/n} \rfloor \times \lfloor 612/\sqrt{750/n} \rfloor$. For each $n \in \{600, 650, \ldots, 900, 950\}$ we randomly generated 50 maps. From each generated map, we selected its largest connected component and used that as the input for our algorithm both with and without SRS. Figure 5.8 shows the average branch-and-bound tree sizes and running times for these experiments. The reported branch-and-bound tree sizes for the case with SRS includes the nodes in branch-and-bound trees of recursive calls. Considering the logarithmic scale on the vertical axis, the potential savings from using SRS are clear.

Comparison

Christensen *et al.* [27] reports on experiments with different heuristic algorithms on problem instances with up to 1500 cities. The running times of the different heuristics as mentioned by Christensen *et al.*, observed on a DEC 3000/400 AXP workstation, fall in the range of tenth of seconds (for random solutions) to approximately 1000 seconds (for the heuristic by Zoraster [128]). The experiments presented in this section clearly show that map labelling instances with up to 950 cities can be solved to optimality using LP-based cutting plane/branch-and-bound and branch-and-cut algorithms within reasonable time. Moreover, they show that rounding algorithms are capable of producing optimal or close-to-optimal solutions. We conclude that our LP-based algorithms are presently among the best for solving map labelling problems.

Chapter 6

The Merchant Subtour Problem

6.1 Introduction

A *merchant* is a person who travels around in a vehicle of fixed capacity, and makes a living by buying goods at places where they are cheap and selling them at places where he can make a profit. A *merchant subtour* is a directed closed walk of a merchant starting and ending at a given place, together with a description of the load of the vehicle along each traversed arc. Since travelling is not for free, we assume the merchant has to pay a cost that is linear in the distance he travels. This chapter addresses the problem of finding a minimum cost merchant subtour where the cost of a merchant subtour equals the travel cost minus the profit made by trading along the way. The Merchant Subtour Problem (MSP) is the problem of finding a minimum cost merchant subtour.

Our interest in the merchant subtour problem was started because it models the pricing problem of the Van Gend & Loos vehicle routing problem that is the subject of Chapter 7. Apart from this, the merchant subtour problem is related to several problems that are reported on in the literature. It has a very nice structure, making it an interesting problem to study in its own right.

6.1.1 Problem Definition

In this section we formalise the MSP. We are given a directed graph $G = (V, A)$, where the set of nodes V corresponds to a set of *distribution centres*, and $A = \{(u, v) \mid u, v \in V\}$ is a set of directed arcs (including self-loops). Furthermore we are given a set of *commodities* K, a vector $\boldsymbol{d} \in \mathbb{N}^K$, $D \in \mathbb{N}$, a vector $\boldsymbol{t} \in \mathbb{N}^A$, $T \in \mathbb{N}$, and a vector $\boldsymbol{c} = (\boldsymbol{c}_A, \boldsymbol{c}_K)$ with $\boldsymbol{c}_A \in \mathbb{N}^A$ and $\boldsymbol{c}_K \in \mathbb{Z}^K$. With every commodity $k \in K$, a unique ordered pair $(u, v) \in V \times V$ is associated, $u \neq v$. Here, u is the *source* and v is the *destination* of the commodity. We will sometimes use $(u, v) \in K$ to denote k (which should not be confused with arc

$(u, v) \in A$). Further, $\boldsymbol{d}$ is the *demand* vector, i.e., for each $k \in K$ the maximum amount of commodity k that can be shipped is d_k. The *capacity* of our vehicle is D. Note that it is not required that the merchant ships all demand. The travel times are given in the vector $\boldsymbol{t}$, i.e., for each $a \in A$, t_a is the time it takes to traverse arc a. T is the maximum amount of time we are allowed to use in total, and $\boldsymbol{c}$ is the cost vector, i.e., for each $a \in A$ the cost of traversing a is c_a. For each $k \in K$ the profit made by shipping k is $-c_k$ per unit. We assume that for all self-loops $a = (v, v) \in A$, $c_a = t_a = 0$ and that $\boldsymbol{c}_K \leq \boldsymbol{0}$. Finally, there is a special node $s \in V$ called the *depot*, that is the starting location of our merchant. We will make one assumption:

Assumption 6.1. *The subtour traversed by the merchant is a directed cycle.*

We can use a network transformation called node duplication (see Ahuja, Magnanti, and Orlin [5]) and slightly generalise the demands in such a way that they are associated with sets of nodes to allow for the situation in which every node of the original problem can be visited a number of times. In the presence of a finite time limit and strictly positive travel times, this is sufficient to allow any possible closed walk to appear as part of a solution. However, we will see that the MSP is $\mathcal{NP}$-hard. This means that from a computational point of view node duplication is not very attractive, as it blows up the dimensions of the problem. Assumption 6.1 does allow us to make the following definition:

Definition 6.1. A *merchant subtour* starting from s is a tuple $(C, \boldsymbol{\ell})$, where $C = (s = v_0, a_1, v_1, \ldots, a_n, v_n = s)$ with $a_i = (v_{i-1}, v_i) \in A$ is a directed cycle in G, and $\boldsymbol{\ell} \in \mathbb{N}^K$ indicates the load of the vehicle.

Figure 6.1 depicts a merchant subtour. We denote the nodes (arcs) of a directed cycle C by $V(C)$ (and $A(C)$, respectively), so with C as above $V(C) = \{v_1, \ldots, v_n\}$ and $A(C) = \{a_1, \ldots, a_n\}$.

Definition 6.2. A merchant subtour $(C, \boldsymbol{\ell})$ starting from s is *feasible* if

(i) for each $k \in K$ with $\ell_k > 0$ we have $k = (v_i, v_j)$ for some $v_i, v_j \in V(C)$ with $i < j$,

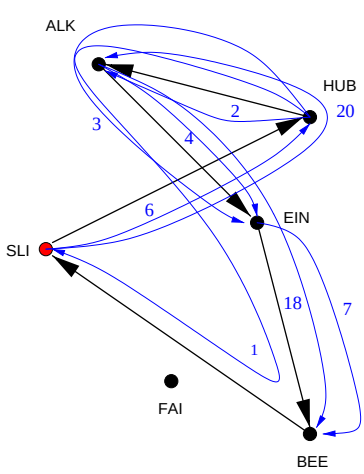
(ii) for each arc $(v_{i-1}, v_i) \in A(C)$ we have that

$$\boldsymbol{\ell}(\{(v_{j_1}, v_{j_2}) \in K \mid j_1 \in \{1, \ldots, i-1\}, j_2 \in \{i, \ldots, n\}\}) \leq D,$$

(iii) $\boldsymbol{\ell} \leq \boldsymbol{d}$, and

(iv) $\boldsymbol{t}(A(C)) \leq T$.

Here (i) enforces that commodities are only moved in the direction of the cycle, (ii) enforces that the total amount of commodities that passes any arc a in this directed cycle is at most the capacity of our vehicle, (iii) enforces that we do not exceed the demand, and (iv) enforces that the maximum travel time is not exceeded.



This figure presents two views on the same merchant subtour. Left: the straight arrows indicate the directed cycle followed by the merchant, that starts and ends in SLI. The curved arrows indicate commodities that are taken along. Numbers indicate quantities. FAI is not visited. Above: the contents of the vehicle over the time after aggregating the commodities per destination, and assigning each destination a different shade.

Figure 6.1: A Merchant Subtour

6.1.2 Related Literature

A merchant subtour is a tour through a directed graph. This relates the merchant subtour problem to the asymmetric travelling salesman problem, reported on, among others, by Grötschel and Padberg [56] and Balas and Fischetti [11]. Moreover, a merchant subtour does not have to include all nodes of the graph, which makes it related to the prize collecting travelling salesman problem, reported on by Balas [9, 10], and the weighted girth problem, reported on by Bauer [15]. The merchant subtour problem differs from the latter two problems in that we have one given node (namely, the depot) that has to be included in the subtour, and in that the cost of a tour is not a linear function of the arcs in the tour, but also incorporates a term that depends on the demand that is shipped.

6.1.3 The Loading Problem

Let

$$C = (s = v_0, (v_0, v_1), v_1, \ldots, (v_{n-1}, v_n), v_n = s) \tag{6.1}$$

be a directed cycle in G. In this section we discuss the merchant subtour problem restricted to the case that the graph $G = (V, A)$ is the directed cycle C, i.e., $V = V(C)$ and $A = A(C)$. In this case, we know the route of the vehicle, and only have to decide on its load. We call such a restricted merchant subtour problem a *loading problem*. To avoid confusion, we will refer to the original merchant subtour problem where G can be an arbitrary directed graph

as the *unrestricted* merchant subtour problem. We will introduce an integer programming formulation that describes the loading problem, and prove that its constraint matrix is totally unimodular (see e.g. Schrijver [104]). This result is fundamental for the algorithms in the remainder of this chapter, as it allows us to generalise local search neighbourhoods for the travelling salesman problem to the unrestricted MSP, and it allows us to give a good integer programming formulation for the unrestricted MSP.

For $(v_i, v_j) \in K$, let $\mathcal{P}^C_{(v_i,v_j)}$ be the set of all paths from v_i to v_j in G that do not have s as internal node. By the numbering of the nodes induced by C and because paths are simple, we have that

$$\mathcal{P}^C_{(v_i,v_j)} = \begin{cases} \{(v_i, (v_i, v_{i+1}), v_{i+1}, \ldots, (v_{j-1}, v_j), v_j)\} & \text{if } i < j, \\ \varnothing, & \text{otherwise.} \end{cases}$$

Let $\mathcal{P}^C = \bigcup_{k \in K} \mathcal{P}^C_k$ be the set of all paths that connect the source and the sink of some commodity ($\mathcal{P}^C$ contains all paths with demand).

We use the decision variables $\boldsymbol{f} \in \mathbb{R}^{\mathcal{P}^C}$, where for each path $P \in \mathcal{P}^C$ with P a path from u to v, f_P indicates the amount of commodity (u, v) we ship. The loading problem can now be stated as the following integer programming problem:

$$\begin{aligned} z^*_{\text{load}}(C) = \min \quad & \boldsymbol{c}_A(A(C)) + \sum_{(u \rightsquigarrow v) \in \mathcal{P}^C} (\boldsymbol{c}_K)_{(u,v)} f_{(u \rightsquigarrow v)} && && \text{(6.2a)} \\ \text{subject to} \quad & \boldsymbol{f}(\{P \in \mathcal{P}^C \mid A(P) \ni a\}) \leq D && \forall a \in A, && \text{(6.2b)} \\ & f_{(u \rightsquigarrow v)} \leq d_{(u,v)} && \forall (u \rightsquigarrow v) \in \mathcal{P}^C, && \text{(6.2c)} \\ & -\boldsymbol{f} \leq \boldsymbol{0}, \text{ integer.} && && \text{(6.2d)} \end{aligned}$$

Here, inequalities (6.2b) enforce that the capacity of the vehicle is not exceeded, and inequalities (6.2c) enforce that we do not exceed the demand. Observe that if $\boldsymbol{f}$ is a feasible solution to model (6.2), then $(C, \boldsymbol{\ell})$ with $\boldsymbol{\ell}_{(u,v)} = f_{(u \rightsquigarrow v)}$ if $(u \rightsquigarrow v) \in \mathcal{P}^C$ and $\boldsymbol{\ell}_{(u,v)} = 0$ otherwise is a feasible merchant subtour of the same value.

Lemma 6.2. *The constraint matrix of model (6.2) is totally unimodular.*

Proof. Let $M \in \{0,1\}^{\mathcal{P}^C \times A}$ be the transpose of the constraint matrix of inequalities (6.2b), i.e.,

$$M = \left(\chi^{A(P)} \mid P \in \mathcal{P}^C\right)^T.$$

We claim that M is totally unimodular. From this claim the lemma follows because with M, also the matrix

$$\begin{pmatrix} M^T \\ I \\ -I \end{pmatrix}$$

is totally unimodular, which is the constraint matrix of model (6.2).

It remains to prove our claim. Choose a subset $A' \subseteq A$ of the arcs arbitrarily. Order the arcs in A' in the order in which they occur on the directed cycle (6.1), i.e., such that $A' = \{a_1, \ldots, a_{|A'|}\}$ where for a_{j_1}, a_{j_2} with $j_1 < j_2$ we have $a_{j_1} = (v_{i_1}, v_{i_1+1}), a_{j_2} = (v_{i_2}, v_{i_2+1})$ with $i_1 < i_2$. Let $A^{\text{odd}} = \{a_j \in A' \mid j \text{ odd}\}$ be the arcs in A' with odd indices, and $A^{\text{even}} = \{a_j \in A' \mid j \text{ even}\}$ be the arcs in A' with even indices. Choose $P = (v_{i_1}, (v_{i_1}, v_{i_1+1}), \ldots, (v_{i_2-1}, v_{i_2}), v_{i_2}) \in \mathcal{P}^C$ arbitrarily. Observe that

$$A(P) \cap A' = \{a_{j_1}, a_{j_1+1}, \ldots, a_{j_2-1}, a_{j_2}\}$$

for some choice of j_1, j_2. Denote by $\boldsymbol{m}_P$ the row of matrix M indexed by P, and recall that $\boldsymbol{m}_P \in \{0,1\}^A$ is the incidence vector of $A(P)$. Hence,

$$\boldsymbol{m}_P(A^{\text{odd}}) - \boldsymbol{m}_P(A^{\text{even}}) = |A^{\text{odd}} \cap A(P)| - |A^{\text{even}} \cap A(P)| \in \{-1, 0, 1\}.$$

Because P and A' were chosen arbitrarily, it follows that each collection of columns of M can be split into two parts so that the sum of the columns in one part minus the sum of the columns in the other part is a vector with entries only $0, +1$, and -1. By Ghouila-Houri [52] (see also Schrijver [104, Theorem 19.3]), this proves our claim and completes the proof of the lemma. □

As a consequence, any basic solution to the LP relaxation of model (6.2) will be integral. Because LP relaxations can be solved in polynomial time [55], and the dimensions of model 6.2 are polynomial in $|V|$, it follows directly from Lemma 6.2 that we can solve the loading problem in polynomial time. Moreover, if we have an (optimal) fractional solution $\boldsymbol{f}$ to model (6.2), we can write it as a convex combination of (optimal) integer solutions to model (6.2). This implies that within the context of the unrestricted merchant subtour problem, we do not need to enforce integrality conditions on the flow of commodities, even when these are required to be integral. Indeed, given a merchant subtour that ships commodities in fractional amounts, we can always find a merchant subtour with the same value that is integral by solving the loading problem on the induced directed cycle of the merchant subtour. After relaxing the integrality conditions on the load shipped, we may assume that the vehicle capacity is $D = 1$, and that all demand is expressed in (fractions of) vehicle loads.

Assumption 6.3. *The vehicle capacity D equals 1.*

This assumption is non-restrictive, as we can always scale a merchant subtour problem by multiplying the demand vector $\boldsymbol{d}$ with a factor $1/D$ and multiplying the cost vector $\boldsymbol{c}_K$ by a factor D without changing the value of the optimal solution. Note that for such a scaled problem the demand vector then is an element from $\frac{1}{D}\mathbb{N}^K$ instead of from $\mathbb{N}^K$, and that feasible merchant subtours are of the form $(C, \boldsymbol{\ell})$ with $\boldsymbol{\ell} \in \frac{1}{D}\mathbb{N}^K$ (we also "scale" Definition 6.1). In the remainder of this chapter, we assume we are working with a scaled problem unless mentioned otherwise.

6.1.4 An Integer Programming Formulation for the MSP

In this section we introduce an integer programming formulation for the merchant subtour problem. For each commodity $(u, v) \in K$, let $\mathcal{P}_{(u,v)}$ denote the set of all possible paths in G from u to v that do not have s as an internal node, and let $\mathcal{P}$ denote the set of all such paths in G connecting the source and destination of some commodity (i.e., $\mathcal{P} = \bigcup_{k \in K} \mathcal{P}_k$). We use the decision variables $\boldsymbol{x} \in \{0,1\}^A$, $\boldsymbol{f} \in \mathbb{R}_+^{\mathcal{P}}$, and $\boldsymbol{z} \in \mathbb{R}_+^K$, where $x_a = 1$ when we use arc a and $x_a = 0$ otherwise. For a path $P = (u \rightsquigarrow v) \in \mathcal{P}$ the variable $\boldsymbol{f}_P$ indicates the amount of commodity (u, v) we ship via path P, and z_k indicates the total amount of commodity k that we ship.

The following formulation is obtained by extending the formulation of the prize collecting travelling salesman problem by Balas [9] with the $\boldsymbol{f}$- and $\boldsymbol{z}$-variables to model the loading dimension of our problem:

$$
\begin{aligned}
\min \quad & z_{\mathrm{MSP}}(\boldsymbol{x}, \boldsymbol{f}, \boldsymbol{z}) = \boldsymbol{c}_A^T \boldsymbol{x} + \boldsymbol{c}_K^T \boldsymbol{z} && && \text{(6.3a)}\\
\text{subject to} \quad & \boldsymbol{x}(\delta^-(v)) + x_{(v,v)} = 1 && \forall v \in V, && \text{(6.3b)}\\
& \boldsymbol{x}(\delta^+(v)) + x_{(v,v)} = 1 && \forall v \in V, && \text{(6.3c)}\\
& \boldsymbol{x}(A(S) \setminus \{(v,v)\}) \leq |S| - 1 && \forall v \in S \subseteq V \setminus \{s\} && \text{(6.3d)}\\
& \boldsymbol{t}^T \boldsymbol{x} \leq T, && && \text{(6.3e)}\\
& \boldsymbol{f}(\mathcal{P}_k) = z_k && \forall k \in K, && \text{(6.3f)}\\
& \boldsymbol{f}(\{P \in \mathcal{P} \mid A(P) \ni a\}) \leq x_a && \forall a \in A, && \text{(6.3g)}\\
& \boldsymbol{z} \leq \boldsymbol{d}, && && \text{(6.3h)}\\
& \boldsymbol{x} \in \{0,1\}^A, \boldsymbol{f} \in \mathbb{R}_+^{\mathcal{P}}, \boldsymbol{z} \in \mathbb{R}_+^K. && && \text{(6.3i)}
\end{aligned}
$$

Here, equalities (6.3b) and (6.3c) are called the *in-* and *out-degree constraints*, respectively. The in- and out-degree constraints enforce that for each node $v \in V$ we either use the self-loop (v, v), or one arc entering and one arc leaving v. Inequalities (6.3d) are the *subtour elimination constraints*. They make solutions containing a subtour that does not include node s infeasible. Inequalities (6.3e) enforce that we do not exceed the time limit. Equalities (6.3f) link the $\boldsymbol{z}$-variables with the $\boldsymbol{f}$-variables. Inequalities (6.3g) are called the *capacity* constraints, and enforce that we only allow flow of commodities on arcs that are in the subtour as indicated by the $\boldsymbol{x}$ variables. Finally, inequalities (6.3h) enforce that we do not exceed the demand. Note that the $\boldsymbol{z}$-variables are redundant and can be eliminated from the model using equality (6.3f).

The following theorem shows that (6.3) is indeed the problem that we want to solve.

Theorem 6.4. *Let $(\boldsymbol{x}, \boldsymbol{f}, \boldsymbol{z})$ be a basic feasible solution to the LP relaxation of (6.3) with $\boldsymbol{x} \in \{0,1\}^A$. Then $D\boldsymbol{f} \in \mathbb{N}^{\mathcal{P}}$ for all $D \in \mathbb{N}$ such that $D\boldsymbol{d} \in \mathbb{N}^K$.*

Proof. Choose $D \in \mathbb{N}$ such that $D\boldsymbol{d} \in \mathbb{N}^K$. By substituting (6.3h) in (6.3f) we

find that $\boldsymbol{f}$ satisfies

$$\begin{aligned} \boldsymbol{f}(\{P \in \mathcal{P} \mid A(P) \ni a\}) &\leq x_a && \forall a \in A && \text{(6.4a)} \\ \boldsymbol{f}(\mathcal{P}_k) &\leq d_k && \forall k \in K, && \text{(6.4b)} \\ \boldsymbol{f} &\geq \mathbf{0} && && \text{(6.4c)} \end{aligned}$$

Since $\boldsymbol{x} \in \{0,1\}^A$ and $\boldsymbol{x}$ satisfies (6.3c)–(6.3d), the set of arcs $A(C) = \text{supp}(\boldsymbol{x})$ induces a directed cycle C in G. It follows that for all paths $P \in \mathcal{P}$ $f_P > 0$ implies $A(P) \subseteq A(C)$, and $A(P) \not\subseteq A(C)$ implies $f_P = 0$. Hence we have $\boldsymbol{f}_{\mathcal{P} \setminus \mathcal{P}^C} = \mathbf{0}$, and

$$\begin{aligned} \boldsymbol{f}(\{P \in \mathcal{P}^C \mid A(P) \ni a\}) &\leq 1 && \forall a \in A(C), && \text{(6.5a)} \\ \boldsymbol{f}(\mathcal{P}_k^C) &\leq d_k && \forall k \in K, && \text{(6.5b)} \\ \boldsymbol{f}_{\mathcal{P}^C} &\geq \mathbf{0}, && && \text{(6.5c)} \end{aligned}$$

where $\mathcal{P}^C, \mathcal{P}_k^C$ are as in Section 6.1.3 (note that $\mathcal{P}_k^C$ is a singleton set). But then, $\boldsymbol{f}' = D\boldsymbol{f}_{\mathcal{P}^C}$ satisfies

$$\begin{aligned} \boldsymbol{f}'(\{P \in \mathcal{P}^C \mid A(P) \ni a\}) &\leq D && \forall a \in A(C), && \text{(6.6a)} \\ f'_{(u \rightsquigarrow v)} &\leq D d_{(u,v)} && \forall (u \rightsquigarrow v)\mathcal{P}^C && \text{(6.6b)} \\ \boldsymbol{f}' &\geq \mathbf{0}. && && \text{(6.6c)} \end{aligned}$$

Because $(\boldsymbol{x}, \boldsymbol{f}, \boldsymbol{z})$ is a basic feasible solution to (6.3), $\boldsymbol{f}'$ is a basic feasible solution to (6.6). By choice of D and Lemma 6.2 we find that $\boldsymbol{f}'$ is integral. Hence $D\boldsymbol{f}$ is integral. □

Proposition 6.5. *Let $\boldsymbol{x}$ be defined as $x_a = 1$ if $a = (v, v)$ for some $v \in V$, and $x_a = 0$ otherwise. Then, $(\boldsymbol{x}, \mathbf{0}, \mathbf{0})$ is a feasible solution and $z_{MSP}(\boldsymbol{x}, \mathbf{0}, \mathbf{0}) = 0$.*

Proof. Recall that $c_{(v,v)} = t_{(v,v)} = 0$ for all $v \in V$. The proof is directly from model (6.3). □

Proposition 6.5 states that it is feasible for the merchant to stay at the depot. In this situation, the merchant doesn't have any cost, but neither does he make a profit, so the objective function evaluates to 0. So if $(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*)$ is the optimal merchant subtour, then $z_{\text{MSP}}(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*) \leq 0$.

6.1.5 The Computational Complexity of the MSP

In this section we show that the merchant subtour problem is $\mathcal{NP}$-hard by a reduction from the travelling salesman problem. We first show that for suitably chosen $\boldsymbol{c}_K$, the optimal solution of the merchant subtour problem maximises the volume of the demand shipped. Using this, given a TSP instance, we define an MSP instance in which the optimal solution contains an optimal travelling salesman tour.

Lemma 6.6. *Let $\bar{z}$ be an upper bound on the travel cost $\boldsymbol{c}_A^T\boldsymbol{x}$ of any subtour $\boldsymbol{x}$. For $k \in K$, let $c_k = -(\bar{z}+1)$. Then, the optimal solution to the MSP maximises $\sum_{k\in K} z_k = \boldsymbol{z}(K)$.*

Proof. The proof is by contradiction. Let $(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*)$ be an optimal solution to the MSP, and suppose there exists a solution $(\boldsymbol{x}', \boldsymbol{f}', \boldsymbol{z}')$ with $\boldsymbol{z}'(K) > \boldsymbol{z}^*(K)$. Then:

$$\begin{aligned} z_{\text{MSP}}(\boldsymbol{x}', \boldsymbol{f}', \boldsymbol{z}') - z_{\text{MSP}}(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*) &= \boldsymbol{c}_A^T\boldsymbol{x}' + \boldsymbol{c}_K^T\boldsymbol{z}' - (\boldsymbol{c}_A^T\boldsymbol{x}^* + \boldsymbol{c}_K^T\boldsymbol{z}^*) \\ &\leq \bar{z} + \boldsymbol{c}_K^T\boldsymbol{z}' - (\boldsymbol{c}_A^T\boldsymbol{x}^* + \boldsymbol{c}_K^T\boldsymbol{z}^*) \\ &\leq \bar{z} + \boldsymbol{c}_K^T\boldsymbol{z}' - \boldsymbol{c}_K^T\boldsymbol{z}^* \\ &= \bar{z} + \sum_{k\in K} c_k z'_k - \sum_{k\in K} c_k z^*_k \\ &= \bar{z} - (\bar{z}+1)(\boldsymbol{z}'(K) - \boldsymbol{z}^*(K)) \\ &< 0. \end{aligned}$$

Hence $z_{\text{MSP}}(\boldsymbol{x}', \boldsymbol{f}', \boldsymbol{z}') < z_{\text{MSP}}(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*)$, contradicting the optimality of the solution $(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*)$. It follows that $(\boldsymbol{x}', \boldsymbol{f}', \boldsymbol{z}')$ does not exist. This implies that $(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*)$ maximises $\boldsymbol{z}(K)$. □

Lemma 6.6 gives us a way to let the merchant visit all cities that occur as the source or the sink of some commodity. Given $G = (V, A)$ and $\boldsymbol{c}_A$, we will use this to construct $K, \boldsymbol{c}_K, \boldsymbol{d}, D, \boldsymbol{t}, T$, such that the optimal merchant subtour to this MSP instance induces an optimal travelling salesman tour in G (with respect to $\boldsymbol{c}_A$).

Theorem 6.7. *The merchant subtour problem is $\mathcal{NP}$-hard.*

Proof. The proof is by a reduction from the general asymmetric travelling salesman problem, which is known to be $\mathcal{NP}$-hard (see Johnson and Papadimitriou [65]). Let $G = (V, A)$ be a complete directed graph (including self-loops), and let $\boldsymbol{c}_A$ be the cost of the arcs in A (with $c_{(v,v)} = 0$ for all $v \in V$). The tuple $(G, \boldsymbol{c}_A)$ defines an instance of the asymmetric travelling salesman problem.

We define an instance of the MSP as follows. Take G and $\boldsymbol{c}_A$ as above. Choose any node $s \in V$ arbitrarily as depot. Let $K = \{(s, v) \mid v \in V \setminus \{s\}\}$ and for $k \in K$, let $d_k = 1$. So, we have a demand of one unit from the depot to each other node in the graph. Let the vehicle capacity be $D = \boldsymbol{d}(K) = |V| - 1$, the total demand. Define the driving times $\boldsymbol{t} = \boldsymbol{0}$, and let $T = 0$ be the maximum allowed driving time. It follows from the choice of $\boldsymbol{t}$ and T that every subtour in G satisfies the time constraint. Finally, let $\bar{z} = |V| \max_{a\in A} c_a$, and for all $k \in K$ choose $c_k = -(\bar{z} + 1)$ as in Lemma 6.6.

We will show the existence of a solution to this MSP instance that ships all demand. Let $\{v_1, \ldots, v_{|V|}\}$ be an ordering of V such that $s = v_1$. Let

$$C = (v_1, (v_1, v_2), v_2, (v_2, v_3), \ldots, v_{|V|-1}, (v_{|V|-1}, v_{|V|}), v_{|V|}, (v_{|V|}, v_1), v_1)$$

be a tour in G visiting all nodes. Then the vector $(\chi^{A(C)}, \chi^{\mathcal{P}^C}, \chi^K)$, with $\mathcal{P}^C$ defined as in Section 6.1.3, is a feasible solution to the merchant subtour problem that ships all demand.

Observe that any merchant subtour consists of at most $|V|$ arcs, so $\bar{z}$ is an upper bound on the travel cost of any merchant subtour. It follows from Lemma 6.6 and the choice of D that the optimal solution to the MSP instance ships all demand (i.e., $z_k^* = 1$ for all $k \in K$). If we choose C in such a way that it follows an optimal tour in G, then

$$(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*) = (\chi^{A(C)}, \chi^{\mathcal{P}^C}, \chi^K)$$

must be an optimal solution to the MSP instance since it ships all demand and minimises $\boldsymbol{c}_A^T \boldsymbol{x}$.

Now let $(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*)$ be any optimal solution to the MSP instance defined above. Because it is optimal it must ship all demand. This implies that the subtour induced by $\text{supp}(\boldsymbol{x}^*)$ visits all nodes of G ($x^*_{(v,v)} = 0$ for all $v \in V$), so $\text{supp}(\boldsymbol{x}^*)$ induces a tour in G. From the optimality of $(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*)$ it follows that $(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*)$ minimises

$$z_{\text{MSP}}(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*) = \boldsymbol{c}_A^T \boldsymbol{x}^* + \boldsymbol{c}_K^T \boldsymbol{z}^* = \boldsymbol{c}_A^T \boldsymbol{x}^* - (\bar{z} + 1)|K|.$$

Because $-(\bar{z}+1)|K|$ does not depend on $(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*)$, we conclude that $\text{supp}(\boldsymbol{x}^*)$ induces a tour in G of minimum cost, which is the optimal solution to the asymmetric travelling salesman problem instance. □

Resuming, in this section we have introduced the merchant subtour problem. We have shown that we can solve a special case, called the loading problem, in polynomial time. We have also shown that unless $\mathcal{P} = \mathcal{NP}$, we cannot expect to find polynomial time algorithms that solve the unrestricted MSP to optimality. The remainder of this chapter is organised as follows. In Section 6.2 we give an algorithm that produces feasible solutions to the merchant subtour problem and runs in pseudo-polynomial time. Section 6.3 presents an exponential time algorithm that solves the merchant subtour problem to optimality. We have performed extensive computational experiments to study the behaviour of the algorithms. These experiments are reported on in Section 6.4.

6.2 A Tabu Search Heuristic

In this section we present a tabu search algorithm for the merchant subtour problem. We follow the ideas presented by Herz, Taillard, and de Werra [60]. Our tabu search algorithm works with a relaxation of model (6.3) that is presented in detail in Section 6.2.1. The algorithm works in iterations and maintains the current solution X, the best feasible solution X^* to model (6.3), and the best feasible solution $\tilde{X}^*$ to the relaxation. Denote the current solution in iteration i by X^i. The initial solution X^1 is given in Section 6.2.2. In iteration i, the algorithm computes the best solution X' in a neighbourhood of X^i, and moves

from X^i to X' by setting $X^{i+1} = X'$, even if X^i is better than X' in terms of objective function value of the relaxation. To avoid cycling of the algorithm, the algorithm maintains a set of forbidden neighbourhood transitions, called *tabu moves*. The tabu moves are stored in a list that is called the *tabu list*. Tabu moves are excluded in the neighbourhood search except if they lead to a solution X' that is better than X^* or $\tilde{X}^*$. When a move is made tabu, it stays tabu for a fixed number κ_1 of iterations. The algorithm enters a so-called *diversification phase* when after a fixed number κ_2 of iterations no change of $(X^*, \tilde{X}^*)$ has occurred. If, after another κ_2 iterations still no change of $(X^*, \tilde{X}^*)$ has occurred it terminates, in which case it returns X^*. In our implementation $(\kappa_1, \kappa_2) = (10, 50)$.

We use the following neighbourhoods: the *node insertion* neighbourhood, the *node deletion* neighbourhood, and the *arc exchange* neighbourhood. The node insertion and deletion neighbourhoods are described in Section 6.2.3. The node insertion neighbourhood is used in the nearest insertion algorithm by Rosenkrantz, Stearns, and Lewis II [99]. Its use as a local search neighbourhood to go from one feasible solution to another is new, as is the node deletion neighbourhood. These neighbourhoods arise because in the merchant subtour problem, we are only interested in a subtour; there is no hard constraint that we visit all nodes as in the travelling salesman problem and the vehicle routing problem. There is a relation with the relocation neighbourhood for the vehicle routing problem as described by Kindervater and Savelsbergh [70]. The arc exchange neighbourhood, described in Section 6.2.4, is a directed version of the traditional 2-Opt neighbourhood for the travelling salesman problem [64, 70].

We try to postpone the termination of the search by using so-called *intensification* and diversification techniques. These are described in Section 6.2.5.

We will see that we can optimise over all these neighbourhoods in polynomial time. Since each move that improves a best known solution does improve it by at least one, and since we only allow a constant number of consecutive non-improving iterations, the tabu search algorithm runs in pseudo-polynomial time.

6.2.1 Relaxation of MSP for Tabu Search

Recall from Section 6.1.3 that we can solve the MSP when the graph G is a directed cycle in polynomial time by solving model (6.2). When we are given a directed cycle X in G with $s \in V(X)$, we can construct a solution $(X, \boldsymbol{\ell})$ to the MSP by solving a loading problem to determine an optimal choice of $\boldsymbol{\ell}$. The function

$$z^*_{\text{load}} : \mathcal{C} \to \mathbb{R} : C \mapsto z^*_{\text{load}}(C)$$

defined by model (6.2), that maps each directed cycle $C \in \mathcal{C}$ to the optimal value of the loading problem defined by C, can be used to determine the value of $(X, \boldsymbol{\ell})$.

The tabu search algorithm works on the following formulation:

$$\begin{aligned} \min \quad & z_{\mathrm{TS}}(X) = z^*_{\mathrm{load}}(X) + \rho(\boldsymbol{t}(X) - T)^+ && \text{(6.7a)} \\ \text{subject to} \quad & X \text{ is a directed cycle in } G \text{ with } s \in V(C), && \text{(6.7b)} \end{aligned}$$

where ρ is a self-adjusting penalty coefficient, and $(\boldsymbol{t}(C)-T)^+$ is the violation of the constraint on the total time (6.3e). Model (6.7) is a combinatorial formulation of a relaxation of the projection of model (6.3) on the $\boldsymbol{x}$-variables. Note that for all feasible solutions $(\boldsymbol{x}, \boldsymbol{f}, \boldsymbol{z})$ we have that $z_{\mathrm{MSP}}(\boldsymbol{x}, \boldsymbol{f}, \boldsymbol{z}) = z_{\mathrm{TS}}(X)$, where X is the directed cycle starting from s with $A(X) = \mathrm{supp}(\boldsymbol{x}) \setminus \{(v, v) \mid v \in V\}$.

The penalty coefficient ρ is treated similarly to the way that is used by Gendreau, Laporte and Séguin [50]. It is initially set equal to $|V|$ and is multiplied every ten iterations by $2^{\lfloor((\alpha+4)/5)\rfloor-1}$, where α is the number of infeasible solutions among the last ten solutions. This way the penalty of exceeding the time constraint is multiplied by two if no solution satisfied the time constraint in the last ten iterations, and divided by two if all those solutions satisfied the time constraint. Intermediate cases yield lesser changes of ρ. The intuition here is to find the right balance between feasible and infeasible solutions.

6.2.2 Initial Solution

Initially, the solution is $X = (s, (s, s), s)$ as in Proposition 6.5. As this solution is feasible, it also implies that we start with $X^* = \tilde{X}^* = (s, (s, s), s)$.

6.2.3 Node Insertion and Deletion Neighbourhoods

Suppose we are given a directed cycle $X = (s = v_0, a_1, v_1, \ldots, a_k, v_k = s)$. For each node $w \in V \setminus V(X)$, let $X_{w,i}$ be the directed cycle starting from s obtained from X by inserting node w directly after node v_i, i.e.,

$$X_{w,i} = (s = v_0, \ldots, v_i, (v_i, w), w, (w, v_{i+1}), v_{i+1}, \ldots, v_k = s),$$

and let $i(w)$ be the index minimising the increase in travel cost over all possibilities of $i \in 0, \ldots, k-1$:

$$i(w) = \arg \min_{i=0,\ldots,k-1} \boldsymbol{c}_A^T(\chi^{A(X_{w,i})} - \chi^{A(X)}).$$

Since we have to reformulate and solve the linear program representing the loading problem for each evaluation of the value of a neighbour, we restrict ourself to inserting w after node $v_{i(w)}$. We will call $X_{w,i(w)}$ the *node insertion neighbour* of X that *adds* w.

Definition 6.3. The *node insertion neighbourhood* of a directed cycle X as above is defined as the set

$$\mathcal{N}_1(X) = \{X_{w,i(w)} \mid w \in V \setminus V(X)\}.$$

We can optimise over the node insertion neighbourhood $\mathcal{N}_1(X)$ by solving $|V \setminus V(X)|$ loading problems.

Now, for $i \in 1, \ldots, k-1$ let X_i be the directed cycle obtained from X by removing node v_i and connecting v_{i-1} to v_{i+1}, i.e.,

$$X_i = (v_0, a_1, \ldots, v_{i-1}, (v_{i-1}, v_{i+1}), v_{i+1}, \ldots, a_k, v_k).$$

We call X_i the *node deletion neighbour* of X that removes v_i.

Definition 6.4. The *node deletion neighbourhood* of a directed cycle X as above is defined as the set

$$\mathcal{N}_2(X) = \{X_i \mid i \in 1, \ldots, k-1\}.$$

We can optimise over the node deletion neighbourhood $\mathcal{N}_2(X)$ by solving $|V(X)| - 1$ loading problems, one for each node in $V(X) \setminus \{s\}$.

It remains to specify the tabu mechanism regarding the node insertion and deletion neighbourhoods. If we move to a neighbour X' of X that adds w, and X' does not improve over X in terms of the objective function, then the node deletion neighbour that removes w is made a tabu move. If X' is the node deletion neighbour of X removing node w, and X' does not improve over X in terms of the objective function, then the node insertion neighbour that adds w becomes tabu.

6.2.4 Arc Exchange Neighbourhood

Suppose we are given a directed cycle $X = (s = v_0, a_1, v_1, \ldots, a_k, v_k = s)$. The arc exchange neighbourhood of X is a set of directed cycles X' that can be obtained from X by replacing two arcs by two other arcs, and reversing the direction of a part of the cycle. For $i, j \in 1, \ldots, k-1$ with $i < j$, let $X_{i,j}$ denote the directed cycle obtained from X by exchanging arcs (v_{i-1}, v_i) and (v_j, v_{j+1}) for the arcs (v_{i-1}, v_j) and (v_i, v_{j+1}), respectively, and reversing the direction of X on the segment from v_i to v_j, i.e.,

$$X_{i,j} = (v_0, a_1, \ldots, v_{i-1}, (v_{i-1}, v_j), v_j, a_j^{-1}, \ldots, a_{i+1}^{-1}, v_i, (v_i, v_{j+1}), v_{j+1}, \ldots, a_k, v_k).$$

We say that $X_{i,j}$ is obtained from X by *reversing* the segment from v_i to v_j.

Definition 6.5. The *arc exchange neighbourhood* of a directed cycle X as above is defined as the set

$$\mathcal{N}_3(X) = \{X_{i,j} \mid i \in 1, \ldots, k-2, j \in i+1, \ldots, k-1\}.$$

We can optimise over the arc exchange neighbourhood by solving $(|V(X)| - 1)(|V(X)| - 2)$ loading problems.

The tabu mechanism with respect to the arc exchange neighbourhood is defined as follows. If we move to a neighbour X' that is obtained from X by reversing the segment from v_i to v_j for some indices i, j, and X' does not improve over X in terms of the objective function, then the arc exchange neighbour that reverses the segment from v_j to v_i becomes tabu.

6.2.5 Intensification and Diversification

To continue the search process, we set our current solution to X^*, and modify the objective function (6.7a) to incorporate a penalty term for frequently used arcs. This modified objective function is of the form

$$\tilde{z}_{\mathrm{TS}}(X) = z^*_{\mathrm{load}}(X) + \rho(\boldsymbol{t}(X) - T)^+ + \boldsymbol{\sigma}^T \chi^X,$$

where in iteration i, $\boldsymbol{\sigma} \in \mathbb{R}^A_+$ is defined by

$$\sigma_a = \frac{i_a p_a}{i^2},$$

i_a is the number of iterations in which arc a was in the current solution, i.e.,

$$i_a = |\{j \mid a \in A(X^j), j \in \{1, \ldots, i\}\}|$$

and p_a is a weight that is determined by previous solutions in which arc a took part, i.e.,

$$p_a = - \sum_{\substack{j \in \{1,\ldots,i-1\} \\ a \in X^j}} \frac{z_{\mathrm{TS}}(X^j)}{|A(X^j)|}.$$

The intuition behind this scheme is that an arc that did not occur in any of the previous solutions will be charged its original cost, whereas an arc that was present in all previous solutions will be charged its part in the average solution value.

As soon as we find an improvement of $(X^*, \tilde{X}^*)$ in terms of the objective function (6.7a), we proceed as normal. If this improvement does not occur within κ_2 iterations we give up and terminate the search.

6.3 A Branch-Price-and-Cut Algorithm

In this section we describe how to solve the MSP to optimality using a branch-price-and-cut algorithm. Section 6.3.1 describes the relaxations that will be solved in each node of the branch-and-bound tree. The relaxations are solved by column generation, for which the pricing algorithm is described in Section 6.3.2. Moreover, we strengthen the relaxations by adding valid inequalities described in Section 6.3.3. Note that the out- and in-degree constraints (6.3b) and (6.3c) are GUB constraints. This enables us to use the strengthened criteria for setting variables based on reduced cost from Section 3.2.3. We try to set more variables by logical implications to speed up the computation in each node of the branch-and-bound tree; the logical implications are described in Section 6.3.4. Section 6.3.5 presents the branching scheme we employ to subdivide the solution space.

6.3.1 LP Master Problems of the MSP

Here we describe the relaxations that will be solved in each iteration of the branch-price-and-cut algorithm. Focus on some iteration. The LP relaxation is uniquely determined by bounds $\boldsymbol{l}, \boldsymbol{u} \in \{0,1\}^A$ on the $\boldsymbol{x}$ variables. These bounds are incorporated in the linear programming relaxation of model (6.3), which leads to the following LP master problems:

$$
\begin{aligned}
\min \quad & z^{\boldsymbol{l},\boldsymbol{u}}_{\mathrm{MSP}}(\boldsymbol{x},\boldsymbol{f},\boldsymbol{z}) = \boldsymbol{c}_A^T\boldsymbol{x} + \boldsymbol{c}_K^T\boldsymbol{z} && && \text{(6.8a)}\\
\text{subject to} \quad & \boldsymbol{x}(\delta^-(v)) + x_{(v,v)} = 1 && \forall v \in V, && \text{(6.8b)}\\
& \boldsymbol{x}(\delta^+(v)) + x_{(v,v)} = 1 && \forall v \in V, && \text{(6.8c)}\\
& \boldsymbol{x}(A(S) \setminus \{(v,v)\}) \leq |S| - 1 && \forall v \in S \subseteq V \setminus \{s\} && \text{(6.8d)}\\
& \boldsymbol{t}^T\boldsymbol{x} \leq T, && && \text{(6.8e)}\\
& \boldsymbol{f}(\mathcal{P}_k) = z_k && \forall k \in K, && \text{(6.8f)}\\
& \boldsymbol{f}(\{P \in \mathcal{P} \mid A(P) \ni a\}) \leq x_a && \forall a \in A, && \text{(6.8g)}\\
& \boldsymbol{z} \leq \boldsymbol{d}, && && \text{(6.8h)}\\
& \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}, && && \text{(6.8i)}\\
& \boldsymbol{x} \in \mathbb{R}^A_+, \boldsymbol{f} \in \mathbb{R}^{\mathcal{P}}_+, \boldsymbol{z} \in \mathbb{R}^K_+. && && \text{(6.8j)}
\end{aligned}
$$

In the root node, we take $\boldsymbol{l} = \boldsymbol{0}$ and $\boldsymbol{u} = \boldsymbol{1}$. The subtour elimination constraints (6.8d), as well as some other classes of valid inequalities for model (6.3), are added to the formulation when they are reported by our separation algorithms. Apart from the exact separation algorithm for finding violated subtour elimination inequalities, we use a heuristic that looks for violated lifted cycle inequalities, and we use our procedure to find maximally violated mod-k inequalities. These are described in more detail in Section 6.3.3.

Associate dual variables $\boldsymbol{\pi}$ and $\boldsymbol{\gamma}$ to the constraints (6.8f) and (6.8g), respectively. Choose $(v,w) \in K$ and let

$$P = (v = v_0, a_1, v_2, \ldots, a_k, v_k = w)$$

be a path from v to w in G with $u_a = 1$ for all $a \in A(P)$. The reduced cost of the variable f_P is $-\pi_{(v,w)} - \boldsymbol{\gamma}(A(P))$. Moreover any feasible solution to the dual linear programming problem of model (6.8) satisfies $\boldsymbol{\gamma} \leq \boldsymbol{0}$.

In each node of the branch-and-bound tree, we solve a restricted version of model (6.8) with all the $\boldsymbol{x}$- and $\boldsymbol{z}$-variables, and a subset of the $\boldsymbol{f}$-variables. This gives us a primal solution $(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*)$, together with an optimal solution to the dual linear program of the restricted model (6.8). We check whether all $\boldsymbol{f}$-variables of model 6.8 that can assume strict positive values satisfy their reduced cost optimality criteria using the pricing algorithm described in Section 6.3.2. When there exist $\boldsymbol{f}$-variables that violate the reduced cost optimality criteria, we add them and re-optimise. Otherwise, we have a proof that $(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*)$ is an optimal solution to model (6.8) in the current node.

6.3.2 The Pricing Problem

Focus on any node of the branch-and-bound tree, and assume we are given vectors $\boldsymbol{\pi} \in \mathbb{R}^K$, and $\boldsymbol{\gamma} \in \mathbb{R}^A$ that are part of an optimal solution to the dual linear program of the restricted model (6.8) in that node. A path $P = (v \rightsquigarrow w) \in \mathcal{P}$ violates its reduced cost criteria if $-\pi_{(v,w)} - \boldsymbol{\gamma}(A(P)) < 0$, and allows positive flow if for all $a \in A(P)$ we have $u_a = 1$. Recall from the definition of $\mathcal{P}$ that no path in $\mathcal{P}$ has s as an internal node. Let $\tilde{G} = (\tilde{V}, \tilde{A})$ be the graph obtained from G by eliminating all arcs with $u_a = 0$, duplicating s to s' and redirecting all arcs that enter s so that they enter s' (the self-loop (s, s) becomes the arc (s, s')), and removing all remaining self-loops in $\tilde{A}$. Define a cost vector $\tilde{\boldsymbol{\gamma}} \in \mathbb{R}^{\tilde{A}}$ by setting $\tilde{\gamma}_{(v,w)} = -\gamma_{(v,w)}$ for all $(v, w) \in \tilde{A}$ with $w \neq s'$, and $\tilde{\gamma}_{(v,s')} = -\gamma_{(v,s)}$ for all $(v, s') \in \tilde{A}$.

Lemma 6.8. *Let an upper bound vector $\boldsymbol{u} \in \{0, 1\}^A$ and vectors $\boldsymbol{\pi} \in \mathbb{R}^K$ and $\boldsymbol{\gamma} \in \mathbb{R}^A$ be given, and let $\tilde{G}$, $\tilde{\boldsymbol{\gamma}}$ be as described above. Then, there exists a path $P = (v \rightsquigarrow w) \in \mathcal{P}$ with $-\pi_{(v,w)} - \boldsymbol{\gamma}(A(P)) < 0$ and $\boldsymbol{u}_{A(P)} = \mathbf{1}$ if and only if there exists a path $\tilde{P}$ in $\tilde{G}$ with $\tilde{\boldsymbol{\gamma}}(A(P)) < \pi_{(v,w)}$.*

Proof. We first show that there is a one-to-one correspondence between the paths in $\mathcal{P}$ and the paths in $\tilde{G}$. Choose $P = (v = v_0, a_1, v_1, \ldots, a_k, v_k = w) \in \mathcal{P}$ with $\boldsymbol{u}_{A(P)} = \mathbf{1}$ arbitrarily. Observe that node s can occur only as an end node of P. If $s \notin V(P)$ or if $u = s$, then take $\tilde{P} = P$. If $s = v$, then take $\tilde{P} = (v_0, a_1, v_1, \ldots, v_{k-1}, (v_{k-1}, s'), s')$. By construction of $\tilde{G}$, we have that $V(\tilde{P}) \subseteq \tilde{V}$ and $A(\tilde{P}) \subseteq \tilde{A}$. So, $\tilde{P}$ is a path in $\tilde{G}$. Using the reverse argument, we can show that for any path $\tilde{P}$ in $\tilde{G}$, there exists a path $P \in \mathcal{P}$ with $\boldsymbol{u}_{A(P)} = \mathbf{1}$.

Now, let $P = (v \rightsquigarrow w)$ be a path in $\mathcal{P}$ with $-\pi_{(v,w)} - \boldsymbol{\gamma}(A(P)) < 0$, and let $\tilde{P}$ be its corresponding path in $\tilde{G}$. By construction of $\tilde{\boldsymbol{\gamma}}$, we have that $\tilde{\boldsymbol{\gamma}}(A(\tilde{P})) = -\boldsymbol{\gamma}(A(P))$. Substituting this in $-\pi_{(v,w)} - \boldsymbol{\gamma}(A(P)) < 0$ and rewriting proves the lemma. □

Recall that $\mathcal{P} = \bigcup_{k \in K} \mathcal{P}_k$. Focus on any commodity $k = (v, w) \in K$. By Lemma 6.8, there exists a path $P \in \mathcal{P}$ with $\boldsymbol{u}_{A(P)} = \mathbf{1}$ such that f_P has negative reduced cost if and only if there exists a path $\tilde{P}$ in $\tilde{G}$ from v to w, or to s' if $w = s$, with $\tilde{\boldsymbol{\gamma}}(A(\tilde{P})) < \pi_k$. When $\boldsymbol{\gamma}$ is part of a feasible solution to the dual linear program to the restricted model (6.8), then $\boldsymbol{\gamma} \leq \mathbf{0}$. It follows that $\tilde{\boldsymbol{\gamma}} \geq \mathbf{0}$. We can use Dijkstra's algorithm [42, 5] on $\tilde{G}$ with arc lengths $\tilde{\boldsymbol{\gamma}}$ to compute a shortest path tree that is directed towards a root node $w \in V$. Using this tree we can solve the pricing problem for all commodities with destination w. So the pricing problem can be solved by running Dijkstra's algorithm $|V|$ times, once for each possible destination. As a consequence, the pricing problem can be solved in $O(|V|(|A| + |V| \log |V|))$ time.

6.3.3 Valid Inequalities for the MSP

In this section we describe the classes of valid inequalities we use to find violated valid inequalities in each node of the branch-and-bound tree. When adding valid

inequalities, we have to take care that we do not alter the reduced cost of the $\boldsymbol{f}$-variables, as this would render our pricing algorithm invalid. Therefore, we have to restrict ourself to inequalities that have zero coefficients for all $\boldsymbol{f}$-variables. As there are no constraints linking the $\boldsymbol{z}$-variables to the $\boldsymbol{x}$-variables directly, we look for violated valid inequalities in the $\boldsymbol{x}$-variables only. So, our goal here is to obtain integer values of the $\boldsymbol{x}$-variables. To reach our goal we try to get a good description of the convex hull incidence vectors of subtours containing s. This is done by adding the valid inequalities that are described below. Second, we will branch on the $\boldsymbol{x}$-variables, which is the subject of Section 6.3.5.

Subtour Elimination and Directed Cut Inequalities

Part of the MSP is computing a subtour including node s. Hence, any solution that contains a subtour that does not contain s is infeasible. The subtour elimination constraints (6.3d) are incorporated in model (6.3) to exclude such infeasible solutions. A subtour elimination constraint is indexed by a tuple (v, S) with $v \in S \subseteq V \setminus \{s\}$. By subtracting the out-degree equalities (6.3b) from the subtour elimination constraint induced by (v, S) for all nodes $u \in S$ and multiplying by minus one we obtain the *directed cut constraints* (see also Balas [9])

$$\boldsymbol{x}(\delta^+(S)) + x_{(v,v)} \geq 1. \tag{6.9}$$

Similarly, by using the in-degree equalities (6.3c) we can derive the inequalities

$$\boldsymbol{x}(\delta^-(S)) + x_{(v,v)} \geq 1. \tag{6.10}$$

By a similar argument, the violation (or the slack) of these inequalities is equal, i.e.,

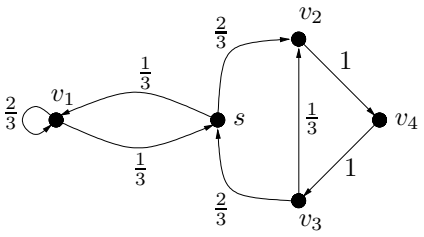
$$\boldsymbol{x}(A(S) \setminus \{(v,v)\}) - |S| + 1 = 1 - \boldsymbol{x}(\delta^+(S)) - x_{(v,v)} = 1 - \boldsymbol{x}(\delta^-(S)) - x_{(v,v)}.$$

Given a fractional solution $(\boldsymbol{x}^*, \boldsymbol{f}^*, \boldsymbol{z}^*)$ to model (6.8), the *support graph* $G(\boldsymbol{x}^*)$ is defined as the directed graph

$$G(\boldsymbol{x}^*) = (V, \text{supp}(\boldsymbol{x}^*)),$$

with arc capacities $\boldsymbol{x}^*$. For each $t \in V \setminus \{s\}$, if the maximum $s - t$ flow in $G(\boldsymbol{x}^*)$ is strictly less than the value $1 - x_{(t,t)}$, then all $s - t$ minimum cuts define a directed cut constraint that is violated by $\boldsymbol{x}^*$. It follows that we can prove that a solution satisfies all directed cut constraints, or we can find a violated directed cut constraint by computing $|V| - 1$ maximum flows. This can be done in $O(|V|^4)$ time.

Observe that because we are interested in computing a subtour, for any $t \in V \setminus \{s\}$ the $s - t$ minimum cut of $G(\boldsymbol{x}^*)$, where $\boldsymbol{x}^*$ is the incidence vector of a subtour, is either zero or one. When $\boldsymbol{x}^*$ is a fractional solution to model (6.8), the $s - t$ minimum cut of $G(\boldsymbol{x}^*)$ can have any value between zero and one. It can occur that there are violated subtour elimination constraints, but none



The global minimum cut is realised by the arc set $\{(s, v_1)\}$, has value $\frac{1}{3}$, and does not correspond to violated subtour elimination constraints. All $s - t$ cuts with $t \in \{v_2, v_3, v_4\}$ have value $\frac{2}{3}$. All subtour elimination constraints on $\{v_2, v_3, v_4\}$ are violated by $\frac{1}{3}$.

Figure 6.2: Global versus $s - t$ Minimum Cut example

of these correspond to a global minimum cut in $G(\boldsymbol{x}^*)$. An example of this is given in Figure 6.2. Padberg and Rinaldi [91] show that the identification of violated subtour elimination constraints for the travelling salesman problem can be done using Gomory and Hu's algorithm [54], for which the best known implementation is Hao and Orlin's algorithm [59]. Gomory and Hu's algorithm only guarantees to report all global minimum cuts. Since we need to compute all the $s - t$ minimum cuts for each $t \in V \setminus \{s\}$, this implies that we cannot use Gomory and Hu's algorithm, nor can we take advantage of clever data structures as presented by Fleischer [45] that are also designed to represent all global minimum cuts only.

Lifted Cycle Inequalities

Given a cycle C in G, a *cycle inequality* has the form

$$\boldsymbol{x}(A(C)) \leq |V(C)| - 1,$$

and states that from all the arcs in the cycle we have to omit at least one in a solution satisfying the inequality. In the MSP, part of the object we are looking for is a directed cycle that includes s. Hence, a cycle inequality obtained from a cycle C in G is valid for the MSP if and only if $s \notin V(C)$. Cycle inequalities are not very strong, in general they can be strengthened by lifting. For the asymmetric travelling salesman problem, this was reported on by Balas and Fischetti [11]. One obvious way to strengthen these inequalities (for the MSP) is add all $\boldsymbol{x}$-variables of the arcs in $A(V(C)) \setminus (A(C) \cup \{(v, v)\})$ with a coefficient of one, for some $v \in V(C)$. This results in the subtour elimination constraint on $(v, V(C))$.

For one special class of lifted cycle inequalities, Balas and Fischetti gave a heuristic that separates them and runs in polynomial time. This class consists of inequalities that are obtained from subtour elimination constraints. In the remainder of this section, we show that this particular class, and its separation algorithm, can be generalised to the MSP.

Proposition 6.9. *Suppose we are given a set $S \subset V \setminus \{s\}$ and $v \in S$. For $q > 1$, let $\{(u_1, w_1), \ldots, (u_q, w_q)\} \subseteq A(S)$ be a collection of distinct arcs and let $\{v_1, \ldots, v_q\} \subseteq V \setminus (S \cup \{s\})$ be a collection of distinct nodes. Let $U = \{u_1, \ldots, u_q\}$, $W = \{w_1, \ldots, w_q\}$, and let $A_q = \bigcup_{i=1}^{q} \{(u_i, v_i), (v_i, w_i), (u_i, w_i)\}$.*

(i) If $v \notin U \cup W$, then the following inequality is valid for model (6.3):

$$\boldsymbol{x}(A(S) \setminus \{(v,v)\}) + \boldsymbol{x}(A_q) \leq |S| + q - 1 \tag{6.11}$$

(ii) If $v \in U \cup W$, then the following inequality is valid for model (6.3):

$$\boldsymbol{x}(A(S)) + \boldsymbol{x}(A_q) \leq |S| + q - 1.$$

Proof. We will first prove (i). Suppose that $v \notin U \cup W$. The proof is by induction on q. For the case that $q = 1$, we add the following inequalities:

$$\boldsymbol{x}(A(S) \setminus \{(v,v)\}) \leq |S| - 1, \tag{6.12}$$
$$\boldsymbol{x}(A(S \cup \{v_1\}) \setminus \{(v_1,v_1)\}) \leq |S|, \text{ and} \tag{6.13}$$
$$2x_{(u_1,w_1)} + x_{(u_1,u_1)} + x_{(w_1,w_1)} + x_{(u_1,v_1)} + x_{(v_1,w_1)} \leq 2. \tag{6.14}$$

This results in the inequality

$$\begin{aligned} &2(\boldsymbol{x}(A(S) \setminus \{(v,v)\}) \\ &\quad + x_{(u_1,w_1)} + x_{(u_1,v_1)} + x_{(v_1,w_1)}) + x_{(u_1,u_1)} + x_{(w_1,w_1)} \\ &\quad + \boldsymbol{x}(A(S \cup \{v_1\}) \setminus ((A(S) \setminus \{(v,v)\}) \cup \{(u_1,v_1),(v_1,w_1)\})) \leq 2|S| + 1. \end{aligned} \tag{6.15}$$

Inequality (6.12) is a subtour elimination constraint on the node set S that is valid for model (6.3) since $s \notin S$. Inequality (6.13) is a subtour elimination constraint on the node set $S \cup \{v_1\}$ that is valid for model (6.3) since $s \notin S \cup \{v_1\}$. Inequality (6.14) is valid for model (6.3) because it follows from the degree constraints and the non-negativity constraints on $\boldsymbol{x}$. It follows that inequality (6.15) is valid for the MSP. After dividing both sides of (6.15) by 2 and rounding the coefficients down, we obtain the valid inequality

$$\boldsymbol{x}(A(S) \setminus \{(v,v)\}) + x_{(u_1,w_1)} + x_{(u_1,v_1)} + x_{(v_1,w_1)} \leq |S|.$$

This is the special case of inequality (6.11) for $q = 1$.

For the induction step let $q = t > 1$, and assume that inequality (6.11) is valid for $q = t - 1$. Adding the inequalities

$$\boldsymbol{x}(A(S) \setminus \{(v,v)\}) + \boldsymbol{x}(A_{q-1}) \leq |S| + q - 2, \tag{6.16}$$
$$\boldsymbol{x}(A(S \cup \{v_q\}) \setminus \{(v_q,v_q)\}) + \boldsymbol{x}(A_{q-1}) \leq |S| + q - 1, \text{ and} \tag{6.17}$$
$$2x_{(u_q,w_q)} + x_{(u_q,u_q)} + x_{(w_q,w_q)} + x_{(u_q,v_q)} + x_{(v_q,w_q)} \leq 2, \tag{6.18}$$

yields inequality (6.11) after dividing by two and rounding down. Because the inequalities (6.16) and (6.17) are valid for model (6.3) by the induction hypothesis, we conclude that inequality (6.11) is valid for model (6.3). This completes the proof of (i).

To prove (ii), observe that if $v \in U \cup W$, then we can assume without loss of generality that $v = u_1$ or $v = w_1$. In this case the same derivation we used to prove clause (i) yields the proof of (ii). □

Suppose we are given a fractional solution $(\boldsymbol{x}^*, \cdot, \cdot)$ to model (6.8). Let $v \in S \subseteq V \setminus \{s\}$ and suppose $\boldsymbol{x}^*(A(S) \setminus \{(v,v)\}) = |S| - 1$. The following is a straightforward adaption of the separation procedure proposed by Balas and Fischetti. Given v, S, and $\boldsymbol{x}^*$ we search for a suitable set of arcs $A_q \subseteq A(S)$ and the associated set of nodes $\{v_1, \dots v_q\} \subseteq V \setminus (S \cup \{s\})$ in a greedy fashion by trying for each arc $(u,w) \in A(S)$ with $v \notin \{u,w\}$ (or with $v \in \{u,w\}$) to identify a node $v' \in V \setminus (S \cup \{s\})$ such that $x_{(u,w)} + x_{(u,v')} + x_{(v',w)} > 1$ (or $x_{(u,w)} + x_{(u,v')} + x_{(v',w)} + x_{(v,v)} > 1$, respectively). Because we start from a tight subtour elimination constraint, this term contributes $x_{(u,w)} + x_{(u,v')} + x_{(v',w)} - 1$ to the violation of the resulting inequality (or $x_{(u,w)} + x_{(u,v')} + x_{(v',w)} + x_{(v,v)} - 1$ in the case that $v \in \{u,w\}$). Therefore, if $A_q \neq \varnothing$, the procedure produces a violated inequality. Moreover, the procedure can be implemented to run in $O(|V|^3)$ time.

Maximally Violated Mod-k Inequalities

We use the separation algorithm described in Section 3.3.5 to find maximally violated mod-k inequalities. The algorithm has as input a set of valid inequalities that are satisfied with equality (*tight*) by the current fractional point, and a value of k. Caprara *et al.* show that it suffices to use only values of k that are prime. In our implementation we run the routine for $k = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29$, which are the 10 smallest prime numbers. Moreover, in order to maximise the opportunity to find maximally violated mod-k cuts, we include as many linearly independent constraints that are satisfied with equality as possible. We include all tight directed cut constraints. These can be derived from the computation of the $s-v$ minimum cuts that was done in order to look for violated directed cut constraints. Next to this, we include all tight constraints that are in the current formulation of model (6.8). These can include non-negativity constraints, and upper bounds of value one on components of $\boldsymbol{x}$ (note that upper bound constraints of value zero can occur but these are only locally valid and therefore excluded), degree constraints (6.3b), (6.3c) (which are tight by definition), the knapsack constraint (6.3e), lifted cycle inequalities (6.11), lifted cover inequalities, and finally mod-k cuts themselves.

Lifted Knapsack Cover Inequalities

Constraint (6.3e) states that we have to pack arcs $a \in A$ that take positive time to complete (namely, t_a) into a working day of length T. Hence, solutions satisfying (6.3e) can be interpreted as the knapsack sets from Section 3.3.3. Hence, lifted cover inequalities derived from (6.3e) are valid for the merchant subtour problem. We use the separation algorithm described in Section 3.3.3 to identify violated lifted cover inequalities.

Combined Separation Algorithms

The separation algorithms described above are combined as follows. Suppose we are given a fractional solution $(\boldsymbol{x}^*, \cdot, \cdot)$ to model (6.8). We start by constructing

the support graph $G(\boldsymbol{x}^*)$. We build a set F of constraints that are satisfied with equality. Initially, F contains all degree constraints, and all non-negativity constraints and upper bounds of one on components of $\boldsymbol{x}$ that are tight in $\boldsymbol{x}^*$. In order to find directed cut constraints that are satisfied with equality or violated, for each node $t \in V \setminus \{s\}$ we compute a maximum $s-t$ flow in $G(\boldsymbol{x}^*)$. Focus on the $s-t$ maximum flow for any $t \in V \setminus \{s\}$, and let ξ be its value.

In the case that $\xi < 1 - x^*_{(t,t)}$ there exist directed cuts separating s from t that correspond to violated directed cut constraints. The *closure* of a set of nodes S in a directed graph is the set of all nodes that occur as the end node of a directed path that starts in S. We compute the smallest set S with $s \in S'$ such that $\delta^+(S)$ is an $s-t$ separating cut; S is the closure of $\{s\}$ in the residual graph of the maximum $s-t$ flow. Observe that $\delta^+(S) = \delta^-(V \setminus S)$, and $s \notin V \setminus S$. Hence the inequality $\boldsymbol{x}(\delta^-(V \setminus S)) + x_{(t,t)} \geq 1$ is a directed cut constraint that is violated by $\boldsymbol{x}^*$ and valid for model (6.3). So, we add $\boldsymbol{x}(\delta^-(V \setminus S)) + x_{(t,t)} \geq 1$ to our formulation.

In the case that $\xi \geq 1 - x^*_{(t,t)}$, all $s-t$ and $t-s$ directed cut constraints are satisfied. As each minimum directed cut that separates s from t corresponds to a tight $s-t$ cut constraint, we compute all $s-t$ minimum cuts and add them to F. Picard and Queranne [95] showed that all $s-t$ minimum cuts appear as closures in the residual graph of the maximum flow that contain s but not t. To enumerate all $s-t$ minimum cuts we first compute all strongly connected components in the residual network of the $s-t$ maximum flow and contract these into single nodes to obtain the *strongly connected component* graph of the residual network. The strongly connected components of the residual network can be computed in $O(|V|+|A|)$ time [29]. Let s' (t') be the strongly connected component that contains s (t', respectively). Note that $s' \neq t'$ because in the residual network of a maximum $s-t$ flow there does not exist any path from s to t. The strongly connected component graph is a directed acyclic graph from which all closures that contain s' but not t' can be enumerated in $O(|V|^3)$ time (see Fleischer [45]). From these closures, we can derive the set of all closures of the residual network that contain s but not t by expanding each node of the strongly connected component graph. Hence, enumerating all tight $s-t$ cuts can be done in $O(|V|^3)$.

We proceed to compute lifted cycle inequalities. To do this, we first construct all sets $S \subset V \setminus \{s\}$ with $t \in S$ such that (t, S) induces a tight subtour elimination constraint, i.e., $\boldsymbol{x}^*(A(S) \setminus \{(t,t)\}) = |S| - 1$, and use these as input for the lifted cycle inequality separation routine described above. We construct all such sets S by computing a $t-s$ maximum flow in $G(\boldsymbol{x}^*)$, and computing all $t-s$ minimum cuts as above. If S_1 is a $s-t$ minimum cut and S_2 is a $t-s$ minimum cut, then the set $S = (V \setminus S_1) \cap S_2$ induces a tight subtour elimination constraint on (t, S). We do this for all combinations of S_1 and S_2, each of them producing a different set S.

Next, we run the separation routine for mod-k cuts starting from the cuts in F. Finally, we run the lifted cover inequality separation on $\boldsymbol{x}^*$.

6.3.4 Logical Implications

In this section we present the logical implications we use to set variables in each node of the branch-and-bound tree. These implications are all on the bounds of the $\boldsymbol{x}$ variables, and force components of $\boldsymbol{x}$ either to 0 or to 1. Note that this is compatible with model (6.8). These implications not only help to speed up the solution of the restricted linear programs, but also make the graph $\tilde{G}$ used in the pricing problem of Section 6.3.2 sparser. This helps in reducing the number of iterations needed for the column generation process to terminate in each node. The implications used are the following:

Proposition 6.10. *Choose any node $v \in V$ arbitrarily, and let $\boldsymbol{x}$ be a feasible solution to model (6.8). Take arc $a \in \delta^+(v) \cup \{(v,v)\}$ (or $a \in \delta^-(v) \cup \{(v,v)\}$), and let $A' = (\delta^+(v) \cup \{(v,v)\}) \setminus \{a\}$ (or $A' = (\delta^-(v) \cup \{(v,v)\}) \setminus \{a\}$, respectively). Then, $l_a = 1$ implies $\boldsymbol{x}_{A'} = \mathbf{0}$, and $\boldsymbol{u}_{A'} = \mathbf{0}$ implies $x_a = 1$.*

Proof. Directly from the degree constraints, the non-negativity constraints, and the upper bound constraints on $\boldsymbol{x}$. □

So, if $x_{(u,v)}$ is set to one, then x_a can be set to zero for all arcs $a \in (\delta^+(u) \cup \{(u,u)\}) \setminus \{(u,v)\}$ and $a \in (\delta^-(v) \cup \{(v,v)\}) \setminus \{(u,v)\}$. If x_a is set to zero for all arcs $a \in (\delta^+(u) \cup \{(u,u)\}) \setminus \{(u,v)\}$ or for all arcs $a \in (\delta^-(v) \cup \{(v,v)\}) \setminus \{(u,v)\}$, then $x_{(u,v)}$ can be set to one. These implications can propagate through the graph. We use the following algorithm to compute as many implications as possible, starting from an initial set of variable settings. Suppose we are give sets $A_0, A_1 \subset A$ of arcs with $\boldsymbol{u}_{A_0} = \mathbf{0}$ and $\boldsymbol{l}_{A_1} = \mathbf{1}$ such that the bounds $\boldsymbol{l}$, $\boldsymbol{u}$ define a feasible instance of model (6.8). For each arc $(u,v) \in A_1$ we compute the set of arcs A' that have $\boldsymbol{u}_{A'} = \mathbf{1}$ and can be set to zero by Proposition 6.10. We set $\boldsymbol{u}_{A'} := \mathbf{0}$, add A' to A_0, and remove (u,v) from A_1. This way, we proceed until A_1 is empty. Next, we process the arcs in A_0. Let arc $(u,v) \in A_0$ and let A' be the set of arcs that have $\boldsymbol{l}_{A'} = \mathbf{0}$ and can be set to one by Proposition 6.10. We remove (u,v) from A_0. If $A' = \varnothing$, we proceed with the next arc. Otherwise, we set $\boldsymbol{l}_{A'} := \mathbf{1}$ and set $A_1 := A'$. This gives us two new sets A_0, A_1, for which we start again from the beginning.

The correctness of this procedure follows from Proposition 6.10. Because every arc is set to 0 or 1 only once, the procedure can be implemented to run in $O(|V| + |A|)$ time.

6.3.5 The Branching Scheme

Here we describe the branching scheme that we use in order to obtain integer solutions. We use a combination of GUB branching and branching on variables. The GUB constraints on which we base the GUB branching scheme are constraints (6.3b) and (6.3c). Next to this, we branch on the $\boldsymbol{x}$ variables. Deciding whether to branch on a GUB or on a variable is done based on pseudo-cost, as described in Section 3.2.5. Observe that whatever branching decision we make, on each branch the LP relaxation differs only in the upper and lower bounds

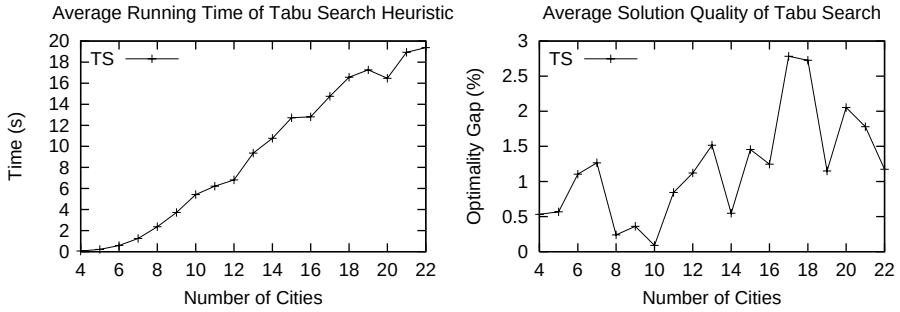


Figure 6.3: Average Running Time and Solution Quality of Tabu Search for the MSP

of the $\boldsymbol{x}$ variables. This implies that the LP relaxations in each node of the branch-and-bound tree are indeed of the form of model (6.8).

6.4 Computational Results

In this section, we present computational results of the algorithms described in this chapter. Since we are interested in the MSP because it arises as subproblem in the context of the Van Gend & Loos problem (see Chapter 7) we used our main instance of the Van Gend & Loos problem to generate random instances for the merchant subtour problem. This was done as follows. The instance of the vehicle routing problem consists of 27 cities, a corresponding distance matrix, and a fixed demand of commodities. For each $n = 4, 5, \ldots, 22$, we generated 50 merchant subtour instances on n cities by selecting 50 distinct subsets of n cities at random, with equal probability. This defines the node set V for each randomly generated problem instance, and we always take $A = V \times V$ as the set of arcs. The arc cost $\boldsymbol{c}_A$ and the demand are taken as in the vehicle routing instance. For each node set V generated this way, we select one node $s \in V$ to be the depot (again with uniform probability). The set K is taken to be the set of all pairs in $V \times V$ for which the vehicle routing problem instance has positive demand. We generate the profits $\boldsymbol{c}_K$ at random such that the expected profit of each commodity equals the distance from its source to its destination, in absolute value, as follows. For $k = (u, v) \in K$, let X be a random variable drawn from a geometric distribution with success probability $p = 1/c_a$, where $a = (u, v) \in A$ is the arc between the source of commodity k and the destination of commodity k. We set $c_k := -X$. The entire test set generated this way consists of 950 instances of the merchant subtour problem.

The solution quality and running times of the tabu search algorithm of Section 6.2 are depicted in Figure 6.3. The running time and branch-and-bound tree sizes of the branch, price and cut algorithm presented in Section 6.3 are depicted in Figure 6.4. The dimensions of the integer program and the constraint

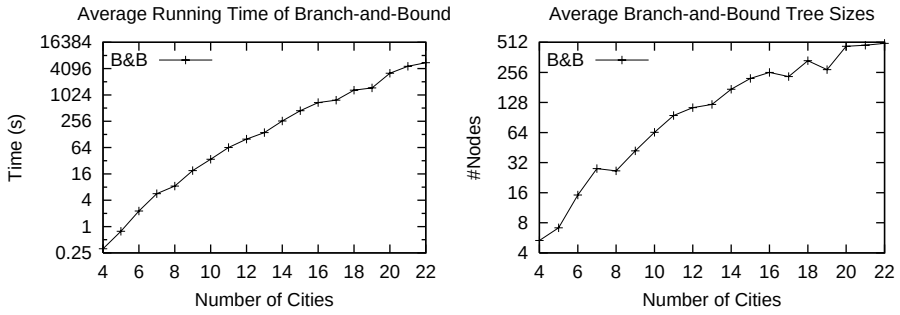


Figure 6.4: Average Branch-and-Bound Running Time and Tree Sizes

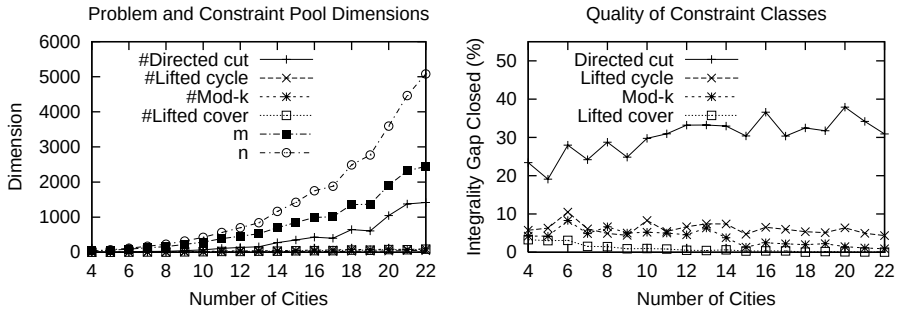


Figure 6.5: Problem Statistics and Constraint Performance

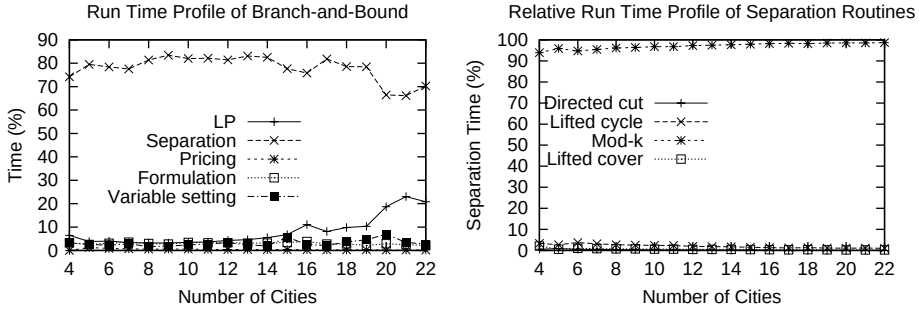


Figure 6.6: Run Time Profiling Data

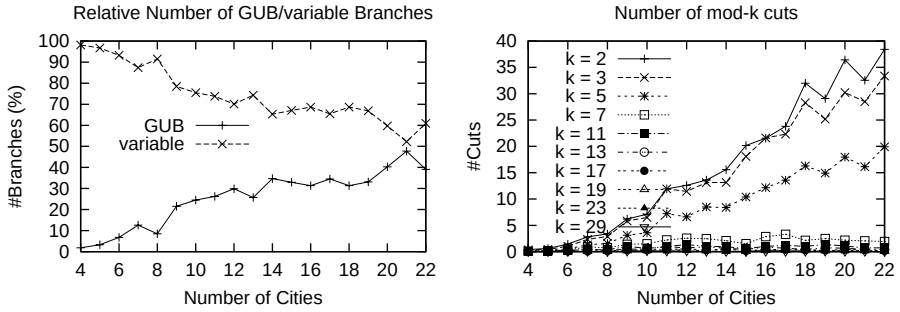


Figure 6.7: Branching Decisions and Mod-k Statistics

pool sizes are presented in Figure 6.5, together with data on the performance of the valid inequalities. The gaps reported in Figure 6.5 are computed as follows. Consider any run of the branch-and-bound algorithm, on any class of cutting planes. Let z_1 be the value of the optimal solution to model (6.8) in the root node without the subtour elimination constraints (the *2-matching relaxation*), let z_2 be the value of the 2-matching relaxation (also in the root node) together with all inequalities of the class of valid inequalities of the class of interest found in this particular run of the algorithm, and let z^* be the value of the optimal integer solution. We measure the gap closed by the class of inequalities by the value $(z_2 - z_1)/(z^* - z_1) * 100\%$. Run time profiles can be found in Figure 6.6. Finally, the mix of variable against GUB branches and the distribution of the mod-k cuts over the different values of k are depicted in Figure 6.7.

From Figure 6.3 it is clear the solutions produced by the tabu search algorithm are of a very high quality. For all problem sizes the reported solutions were within 3% of optimality on average. The running time of the tabu search algorithm behaves equally well.

Our branch-price-and-cut algorithm solves problem instances with up to 22 cities 2 hours of computing time on average. The average branch-and-bound tree sizes on these runs stay just under 512 (see Figure 6.4).

From Figure 6.5 it is clear that the lifted cycle inequalities contribute to the quality of our LP formulations. The same holds for the mod-k inequalities. The lifted cover inequalities appear to be less important. It would be interesting to see whether the so-called *lifted GUB cover inequalities* by Wolsey [124], that can also be used for the MSP, perform better and if so, by how much.

Also from Figure 6.5, and from Figure 6.6, we can see that solving the LP relaxations becomes a substantial part of the running time when the number of variables increases. Applying a primal-dual algorithm (see for example [94]) should allow us to solve the linear programs more efficiently.

The running time within the separation algorithms is used mainly for mod-k separation. This may be due to our policy to include as many tight cuts as possible. Caprara *et al.* [24] show that for the travelling salesman problem such a policy is not necessary. Their ideas can also be applied to the merchant

subtour problem. Again, as already noted in Section 3.3.5 we do not re-use the factorisations of previous runs of the mod-k separation. Doing so might also yield a considerable improvement.

Finally, Figure 6.7 shows that the our GUB branching scheme produces a constant fraction of all branches that are actually performed, and is more competitive for larger values of $|V|$. We conclude that we are able to solve the merchant subtour problem to optimality for real-life instances with up to 22 cities within reasonable time, and we expect that we can solve even larger problems by taking the above considerations into account.

Chapter 7

The Van Gend & Loos Problem

7.1 Introduction

The following distribution problem arises at Van Gend & Loos, a logistics company in the Netherlands. The clients of Van Gend & Loos want to send (small) packages from one place to another. This is organised in the following way. During daytime, the packages are collected at regional distribution centres. This is done using small trucks and vans. Afterwards, in each distribution centre the packages are sorted according to their destination and packages with the same destination are combined in pallets and boxes. After the sorting, each pallet is shipped to a distribution centre that is located near the destination of the packages it contains. This transportation of pallets between distribution centres is done during the night using large trucks. Van Gend & Loos has a fleet of large trucks to accomplish the transportation of pallets between distribution centres during the night. Each truck is stationed at its own distribution centre, drives a route (also called a *trip*) starting from this distribution centre along a number of distribution centres, and returns to its own distribution centre at the end of the night. At each distribution centre along the way it is allowed to load and unload pallets. This loading and unloading has to be done in such a way that all pallets are delivered to the distribution centre that is located near the destination of the packages it contains. The next morning, the packages are delivered from these distribution centres to their destination.

In addition, there are some practical aspects that have to be taken into consideration. These aspects include restrictions on consecutive driving time for each truck, and incorporating time for loading and unloading of trucks. Also, a limited number of the orders have time windows on them.

The subject of this chapter is to develop algorithms that can compute how to do the transportation of the pallets at night. As this is the only problem that arises at Van Gend & Loos that we study in this thesis, we refer to it as the

Van Gend & Loos problem. We focus on a basic abstraction of the problem in which we ignore the additional practical aspects, and the fact that pallets are produced in an on-line fashion by the sorting process.

7.1.1 Problem Definition

The Van Gend & Loos (VGL) problem is defined as follows. We are given a complete directed graph $G = (V, A)$. The set of nodes V corresponds to a set of *distribution centres*. Furthermore, we are given a set of *commodities* K, a set of *trucks* W, vectors $\boldsymbol{c}, \boldsymbol{t} \in \mathbb{N}^A$, a vector $\boldsymbol{d} \in \mathbb{N}^K$, and integers D and T. With each commodity $k \in K$, a unique ordered pair $(u, v) \in V \times V$ is associated, $u \neq v$, where u is the *source* of the commodity and v the *destination*. We will sometimes use $(u, v) \in K$ to denote k. (Note that (u, v) will be used both to denote arc $(u, v) \in A$ and commodity $(u, v) \in K$. The correct interpretation should be clear from the context.) Commodities can only be transported in integral amounts. Each vehicle $w \in W$ is stationed at some distribution centre $v \in V$, which we will denote by $v(w)$, and has a fixed capacity D. Node $v(w)$ is called the *depot* of truck w. The set of all depots is denoted by $v(W) = \bigcup_{w \in W} \{v(w)\}$, and for each $v \in v(W)$ the set of trucks stationed at v is denoted by $W(v) = \{w \in W \mid v = v(w)\}$. The *demand* is given by the vector $\boldsymbol{d}$, i.e., for each $k \in K$ we have to transport precisely d_k units from the source to the destination of k. The *cost* we have to pay when a truck traverses arc a is given by c_a, and the *time* it takes for a truck to traverse arc a is given by t_a. The maximum time a truck may be away from the depot is T. As with the MSP, we make the following assumption:

Assumption 7.1. *The route followed by each truck is a directed cycle.*

Assumption 7.1 will allow us to use the algorithms from Chapter 6. The assumption can be dropped, but this would force us to use generalised versions of the algorithms presented in Chapter 6. Recall the definition of a merchant subtour from Section (6.1.1).

Definition 7.1. A *trip* r is a triple $r = (w^r, C^r, \boldsymbol{\ell}^r)$, where $w^r \in W$ is a truck and $(C^r, \boldsymbol{\ell}^r)$ is a merchant subtour starting from $v(w^r)$.

Definition 7.2. A trip $r = (w^r, C^r, \boldsymbol{\ell}^r)$ is *feasible* if $(C^r, \boldsymbol{\ell}^r)$ is a feasible merchant subtour starting from $v(w^r)$. The *cost* of a trip r is the total cost $\boldsymbol{c}(A(C^r))$ of the arcs in $A(C^r)$. Trip r *covers* $\boldsymbol{\ell}^r$ of the demand and *uses* truck w^r.

Note that Definition 7.2 implicitly involves $\boldsymbol{d}, D, \boldsymbol{t}$, and T.

Definition 7.3. A *solution* to the VGL problem is a set of feasible trips in which each available truck is used in at most one trip.

Definition 7.4. A solution to the VGL problem *covers* the sum of the demands that is covered by the trips in it, and is *feasible* if it covers the demand $\boldsymbol{d}$. The *cost* of a solution to the VGL problem is the sum of the costs of the trips in it.

The instance consists of the demand that is shown in the matrix to the right, and four trucks, stationed at AME, ROT, and two at EIN. Distances and travel times are omitted. Below, depicted are the routes and the loads of the trucks in an optimal solution (see Figure 6.1 for more explanation). The solution was computed in 2:45h by the algorithm treated in Section 7.2.6. The optimal solution was found after four iterations.

	UTR	AME	GRA	EIN	HER	HUB	AMS	SLI	ROO	ROT
UTR	0	24	23	0	0	0	0	23	0	23
AME	0	0	0	0	25	0	0	0	0	26
GRA	0	0	0	0	0	1	49	0	0	50
EIN	0	0	0	0	50	2	0	0	0	0
HER	46	26	0	0	0	4	26	0	0	0
HUB	0	1	2	0	1	0	1	1	3	2
AMS	15	0	26	0	0	9	0	0	26	0
SLI	11	0	0	0	0	6	0	0	24	0
ROO	0	0	0	52	26	9	0	17	0	0
ROT	0	0	49	0	0	1	0	0	25	0

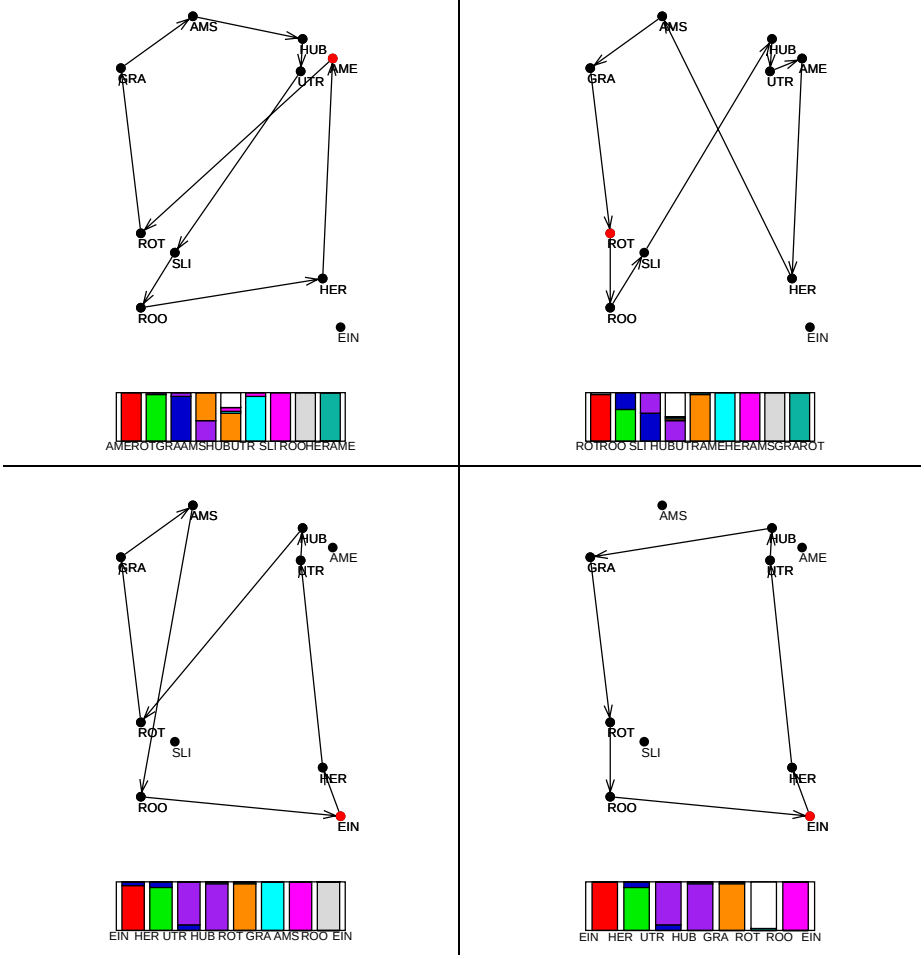


Figure 7.1: An Instance to the VGL Problem and Its Optimal Solution

As example, an instance of the VGL problem and its optimal solution are depicted in Figure 7.1. Instances to the VGL problem can be represented by 8-tuples $(G, \boldsymbol{c}, K, W, \boldsymbol{d}, D, \boldsymbol{t}, T)$, with $G = (V, A)$ as above. The VGL problem is to cover the demand $\boldsymbol{d}$ by a set of feasible trips, using each available truck in at most one of those trips, at minimum total cost. We are primarily interested in solutions that precisely cover (partition) the demand. However, it is easily seen that we can relax this to solutions that cover at least the demand, as follows. If we have a solution to the VGL problem that covers more than $\boldsymbol{d}$ we can find a solution that covers precisely $\boldsymbol{d}$ by just removing the surplus from the load of some trips that are part of the solution. Because this does not alter the cost of the individual trips, this operation does not alter the cost of the entire solution. It is believed that formulations that cover at least the demand are more suitable for algorithmic design than formulations that partition the demand.

The remainder of this chapter is organised as follows. In Section 7.1.2 we will study the computational complexity of the VGL problem. It will turn out that the VGL problem is $\mathcal{NP}$-hard. In Section 7.1.3 we discuss how the VGL problem relates to problems that have been reported on in the literature. Next, we will give an integer programming formulation that forms the basis of our algorithms in Section 7.1.4. In Section 7.2 we present heuristics for the VGL problem that are based on column generation. Finally, we give a branch-and-price algorithm for solving it in Section 7.3. Computational results with our heuristic algorithms are reported on in Section 7.4.

7.1.2 Computational Complexity of The VGL Problem

In this section we study the computational complexity of the VGL problem. We will show that solving the VGL problem is $\mathcal{NP}$-hard, and that deciding whether there exists a feasible solution to the VGL problem is $\mathcal{NP}$-complete. The proof is by a reduction that is similar to the one used in the proof of Theorem 6.7.

Theorem 7.2. *The VGL problem is $\mathcal{NP}$-hard.*

Proof. The proof is by a reduction from the general asymmetric travelling salesman problem, which is known to be $\mathcal{NP}$-hard (see Johnson and Papadimitriou [65]). Let $G = (V, A)$ be a complete directed graph, and let $\boldsymbol{c} \in \mathbb{N}^A$ be the cost vector. The tuple $(G, \boldsymbol{c})$ defines an instance of the asymmetric travelling salesman problem.

We define the following instance of the VGL problem. Take G and $\boldsymbol{c}$ as above. Let the set of trucks be a singleton set $W = \{w\}$ consisting of truck w, and choose its depot $v(w) \in V$ arbitrarily. Let $K = \{(v(w), v) \mid v \in V \setminus \{v(w)\}\}$ and for $k \in K$, let $d_k = 1$. So, we have a demand of one unit from the depot of truck w to each other node in the graph. Let the vehicle capacity be $D = \boldsymbol{d}(K) = |V| - 1$, the total demand. Define the driving times $\boldsymbol{t} = \boldsymbol{0}$, and let $T = 0$ be the maximum allowed driving time. It follows from the choice of $\boldsymbol{t}$ and T that every tour in G satisfies the time constraint.

Observe that for each tour C in G, the singleton set of trips $\{(w, C, \chi^K)\}$ is a solution to the VGL problem instance above of the same cost. Moreover, this

solution is optimal if and only if C is an optimal tour.

Suppose we have an optimal solution to the instance of the VGL problem defined above. Since $|W| = 1$ and $\boldsymbol{d} \neq \boldsymbol{0}$ this solution has the form $\{(w, C^*, \boldsymbol{\ell}^*)\}$. From feasibility of the optimal solution we have $\boldsymbol{\ell}^* \geq \boldsymbol{d}$. Because $(C^*, \boldsymbol{\ell}^*)$ is a feasible merchant subtour and $\ell_k^* \geq d_k = 1$ for all $k \in K$, the directed cycle C^* starts and ends at $v(w)$ and visits all nodes in $V \setminus \{v(w)\}$ on its way, so C^* is a tour. Hence C^* is an optimal tour. □

Theorem 7.3. *The problem of deciding whether there exists a feasible solution to the VGL problem is $\mathcal{NP}$-complete.*

Proof. Observe that the VGL problem is in $\mathcal{NP}$ because writing down a potential solution, checking feasibility and verifying the objective function can all be done in polynomial time. The proof of $\mathcal{NP}$-hardness is again using a reduction from the general asymmetric travelling salesman problem. Given an instance of the asymmetric travelling salesman problem $(G, \boldsymbol{c})$ and a parameter ξ, we construct an instance $(G, \boldsymbol{c}, K, W, \boldsymbol{d}, D, \boldsymbol{t}, T)$ as in the proof of Theorem 7.2, except that this time we take $\boldsymbol{t} = \boldsymbol{c}$ and $T = \xi$.

We will show that there is a bijection between tours of length at most ξ and feasible solutions to the above instance of the VGL problem. Following the lines of the proof of Theorem 7.2, we find that any feasible solution to the instance of the VGL problem gives us a tour of length at most ξ by the constraint on the duration of trips.

Conversely, let τ be a tour of length at most ξ. Let $r = (w^r, C^r, \boldsymbol{\ell}^r)$ be the trip defined by taking $w^r = w$, making C^r by following τ starting from $v(w)$, and taking $\boldsymbol{\ell}^r = \chi^K$. Note that r is a feasible trip. Hence, the singleton set of trips $\{r\}$ is a feasible solution to the instance of the VGL problem. So, there exists a tour of length at most ξ if and only if the instance $(G, \boldsymbol{c}, K, W, \boldsymbol{d}, D, \boldsymbol{t}, T)$ of the VGL problem has a feasible solution. □

7.1.3 Related Literature

The VGL problem can be qualified as a multi-vehicle pickup and delivery problem with multiple depots and goods that can be split in integral amounts. This variant of the vehicle routing problem has, to the best of our knowledge, not been reported on in the literature.

In the special case that the vehicle capacity is 1 the problem coincides with the vehicle routing problem with full loads that is reported on by Desrosiers, Laporte, Sauvé, Soumis, and Taillefer [39].

The VGL problem that we study is related to the problem reported on by Savelsbergh and Sol [103], that also arises at Van Gend & Loos bv. The service of Van Gend & Loos bv can be roughly divided into two parts, the *regular trasportation system* and the *direct transportation system*. The problem described in Section 7.1.1 is an abstraction of a part of the regular transportation system.

It is the direct transportation system that has been studied by Savelsbergh and Sol. There is a fundamental aspect in which the direct transportation system differs from the part of the regular transportation system that we study. Namely, in the problem that we study the demands exceed the vehicle capacity and goods of the same commodity can be split in integral amounts for transportation. In the direct transportation system studied by Savelsbergh and Sol goods fit in the trucks and are kept together for transportation.

General pickup and delivery problems are surveyed by Savelsbergh and Sol [102]. Furthermore, the VGL problem is related to the classical vehicle routing problem (see Laporte and Nobert [75] and Desrochers, Desrosiers, and Solomon [38]), the multi-depot vehicle routing problem (Laporte, Nobert, and Taillefer [76]), the multi-depot vehicle scheduling problem (Ribeiro and Soumis [98], and Carpaneto, Dell'Amico, Fischetti, and Toth [25]), and to the vehicle routing problem with (fractionally) split deliveries (Dror, Laporte, and Trudeau [44]). The vehicle routing problem with fractionally split deliveries that is described by Dror *et al.* resembles the single-depot case of our model, with the integrality constraints on the flow of commodities relaxed. Also, the problem described by Dror *et al.* differs from ours in that they do not consider a pickup and delivery problem, but a problem in which there are only deliveries, i.e., in which all goods are picked up at the depot.

7.1.4 Formulation as a Generalised Set Covering Problem

Here we give a mathematical programming formulation for the VGL problem. The formulation is inspired by the set covering formulation of the vehicle routing problem by Desrochers, Desrosiers and Solomon [38].

Before we formulate the VGL problem as an integer linear programming problem, we introduce the following notion. A *normalised* feasible solution to the VGL problem is a feasible solution to the VGL problem in which all vehicles are assigned to a trip. Given a feasible solution to the VGL problem in which there exist trucks that are not assigned to a trip, we can turn it into a normalised solution by assigning each of those trucks a trivial trip that traverses a self-loop at its depot and does not ship any demand. This does not change the cost of the solution since the cost of traversing self-loops is zero.

Suppose for each truck $w \in W$ we are given the set R_w of all its feasible trips, and let $R = \bigcup_{w \in W} R_w$ be all feasible trips. In order to formulate the VGL problem as an integer linear programming problem, we use the decision variables $\boldsymbol{\lambda} \in \{0,1\}^R$, where for each trip $r \in R$

$$\lambda_r = \begin{cases} 1, & \text{if } r \text{ occurs in the solution, and} \\ 0, & \text{otherwise.} \end{cases}$$

Using these variables, we can formulate the VGL problem as follows:

$$z^*_{\text{VGL}} = \min \quad z_{\text{VGL}}(\boldsymbol{\lambda}) = \sum_{r \in R} \boldsymbol{c}(A(C^r))\lambda_r \tag{7.1a}$$

$$\text{subject to} \quad \sum_{w \in W} \sum_{r \in R_w} \boldsymbol{\ell}^r \lambda_r \geq \boldsymbol{d}, \tag{7.1b}$$

$$\boldsymbol{\lambda}(R_w) = 1 \qquad \forall w \in W, \tag{7.1c}$$

$$\boldsymbol{\lambda} \geq \mathbf{0}, \quad \text{integer}. \tag{7.1d}$$

Let $\boldsymbol{\lambda} = \chi^Q$ be a feasible solution to model (7.1). The inequality constraints (7.1b) enforce that the demand vector $\boldsymbol{d}$ is covered by the trips in Q, and the equality constraints (7.1c) enforce that each truck is used in exactly one trip in Q. Hence, Q is a normalised feasible solution to the VGL problem.

7.2 Heuristics for The VGL Problem

We proceed by presenting heuristic algorithms for the VGL problem. A solution to an instance of the VGL problem is an assignment of trucks to merchant subtours. For the purpose of constructing primal heuristics, this can be exploited by representing solutions to the VGL problem as feasible flows in a special graph that we call the *assignment graph*, which we present in Section 7.2.1. An LP rounding algorithm that exploits the assignment graph is the subject of Section 7.2.2. Once we have a feasible solution, we can try to improve it using local search algorithms. In Sections 7.2.3–7.2.5 we show how to search through local search neighbourhoods by manipulating a flow in the assignment graph. Finally, in Section 7.2.6 we give a revised LP rounding algorithm that combines all the algorithms above.

7.2.1 The Assignment Graph

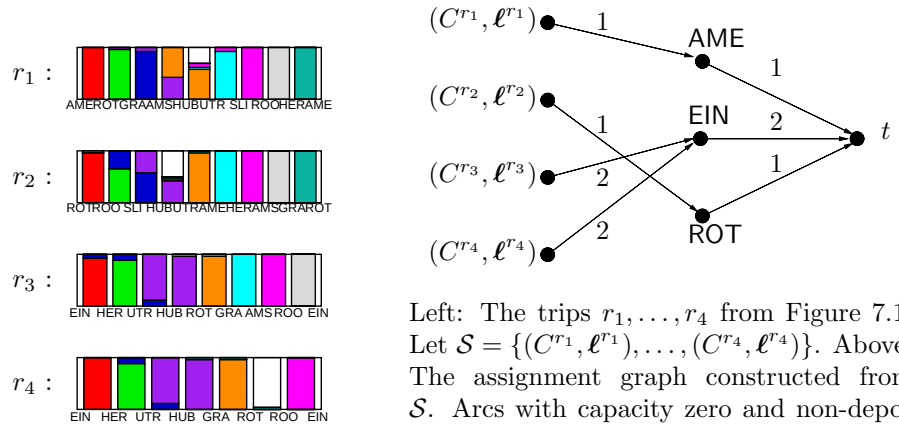
Suppose we are given a set

$$\mathcal{S} = \{(C, \boldsymbol{\ell}) \mid (C, \boldsymbol{\ell}) \text{ is a feasible merchant subtour, } C \text{ starts from a depot}\}.$$

Given a merchant subtour $(C, \boldsymbol{\ell})$ and an arc $a \in A(C)$, let $\boldsymbol{\ell}(a)$ be the load of the vehicle when the merchant traverses a. Consider any node $v \in V(C)$, let C' be the directed cycle starting and ending at v with $V(C') = V(C)$ and $A(C') = A(C)$, and let $u, w \in V(C)$ be the nodes preceding and succeeding v in C, respectively (so $(u, v), (v, w) \in A(C)$). Observe that $(C', \boldsymbol{\ell})$ is a feasible merchant subtour if v has the property that $\boldsymbol{\ell}(u, v)^T \boldsymbol{\ell}(v, w) = 0$ (no commodities "pass" node v). We call all nodes $v \in V(C)$ that satisfy this property *feasible depots* for $(C, \boldsymbol{\ell})$, and denote the set of all feasible depots for $(C, \boldsymbol{\ell})$ by $S(C, \boldsymbol{\ell})$.

Definition 7.5. The *assignment graph* of a VGL problem is the directed acyclic graph $G' = (V', A')$ with node and arc set

$$V' = \mathcal{S} \cup V \cup \{t\}, \text{ and}$$
$$A' = \{(u, v) \mid u \in \mathcal{S}, v \in S(u)\} \cup \{(v, t) \mid v \in V\},$$



Left: The trips $r_1, \ldots, r_4$ from Figure 7.1. Let $\mathcal{S} = \{(C^{r_1}, \ell^{r_1}), \ldots, (C^{r_4}, \ell^{r_4})\}$. Above: The assignment graph constructed from $\mathcal{S}$. Arcs with capacity zero and non-depot nodes have been left out.

Figure 7.2: The Assignment Graph

and associated arc capacities $\boldsymbol{u} \in \mathbb{N}^{A'}$, where $u_{(u,v)} = u_{(v,t)} = |W(v)|$ for all nodes u, v with $v \in V$ and $u \in \mathcal{S}$.

So, for each merchant subtour and for each depot there is a corresponding node in the assignment graph. Nodes in the assignment graph that correspond to merchant subtours are connected by an arc to their feasible depots, and nodes that correspond to depots are connected by an arc to a special sink node t. The capacities are chosen in such a way that the flow from a depot node v to the sink node cannot exceed the total number of trucks stationed at v, and the capacities at arcs leaving nodes that correspond to merchant subtours are chosen in such a way that they do not restrict the value of the flow on those arcs. Note that nodes $v \in V'$ with $W(v) = \varnothing$ can be removed from G', together with all arcs $a \in A'$ with $u_a = 0$. An example of an assignment graph is given in Figure 7.2. Let G' be an assignment graph as defined.

Definition 7.6. A *flow* in G' is a vector $\boldsymbol{x}$ in $\mathbb{R}^{A'}$. A flow $\boldsymbol{x}$ in G' is *feasible* if $\mathbf{0} \leq \boldsymbol{x} \leq \boldsymbol{u}$, and for all $v \in V'$

$$\boldsymbol{x}(\delta^-(v)) - \boldsymbol{x}(\delta^+(v)) \begin{cases} \leq 0 & \text{if } v \in \mathcal{S}, \\ = 0 & \text{if } v \in V, \\ \geq 0 & \text{if } v = t. \end{cases}$$

A feasible flow satisfies flow conservation constraints in all depot nodes, together with non-negativity and upper bound constraints on the arcs. The following propositions state that we can map each solution of an instance of the VGL problem to a feasible flow in G' and vice versa.

Proposition 7.4. *Let $\boldsymbol{\lambda} = \chi^Q$ for some solution $Q \subseteq R$ to the VGL problem*

with $\{(C^r, \boldsymbol{\ell}^r) \mid (w^r, C^r, \boldsymbol{\ell}^r) \in Q\} \subseteq \mathcal{S}$. *Then the flow* $\boldsymbol{x}(\boldsymbol{\lambda})$ *defined by*

$$x_{(u,v)}(\boldsymbol{\lambda}) = \begin{cases} |\{r \in Q \mid (C^r, \boldsymbol{\ell}^r) = u, v(w^r) = v\}|, & \text{if } u \in \mathcal{S} \text{ and } v \in S(u), \\ |\{r \in Q \mid v(w^r) = u\}|, & \text{if } u \in V \text{ and } v = t, \end{cases}$$

is a feasible integer flow in G'.

Proof. Clearly $\boldsymbol{x}(\boldsymbol{\lambda})$ is integral and non-negative. It remains to prove that $\boldsymbol{x}(\boldsymbol{\lambda})$ satisfies the upper bound and flow conservation constraints. We will first show that $\boldsymbol{x}(\boldsymbol{\lambda})$ satisfies the flow conservation constraints. Choose node $v \in V'$ arbitrarily. If $v \in \mathcal{S}$ or $v = t$ the sign of the flow balance follows directly from the non-negativity of $\boldsymbol{x}$. So suppose $v \in V$. By the construction of G', $\{u \mid (u, v) \in \delta^-(v)\} = \{u \in \mathcal{S} \mid S(u) \ni v\}$ and $\delta^+(v) = \{(v, t)\}$. Hence,

$$\begin{aligned} \boldsymbol{x}(\delta^-(v)) = \boldsymbol{x}(\{u \in \mathcal{S} \mid S(u) \ni v\}) &= \sum_{(C,\boldsymbol{\ell}) \in \mathcal{S}: S(C,\boldsymbol{\ell}) \ni v} x_{((C,\boldsymbol{\ell}),v)} \\ &= \sum_{(C,\boldsymbol{\ell}) \in \mathcal{S}: S(C,\boldsymbol{\ell}) \ni v} |\{r \in Q \mid r = (w^r, C, \boldsymbol{\ell}), v(w^r) = v\}| \\ &= \Big| \bigcup_{(C,\boldsymbol{\ell}) \in \mathcal{S}: S(C,\boldsymbol{\ell}) \ni v} \{r \in Q \mid r = (w^r, C, \boldsymbol{\ell}), v(w^r) = v\} \Big| \\ &= |\{r \in Q \mid v(w^r) = v\}| \quad = x_{(v,t)} \\ &= \boldsymbol{x}(\delta^+(v)). \end{aligned}$$

This proves that $\boldsymbol{x}(\boldsymbol{\lambda})$ satisfies the flow conservation constraint in v. Because Q is a solution to the VGL problem, each truck in $W(v)$ is used in at most one trip. It follows that $x_{(v,t)} \leq u_{(v,t)}$. Together with flow conservation, this implies that $\boldsymbol{x}(\boldsymbol{\lambda})$ satisfies the upper bound constraints on all arcs in $\delta^-(v) \cup \delta^+(v)$. Because v was chosen arbitrarily, this completes the proof of the proposition. □

Proposition 7.5. *Let* $\boldsymbol{x}$ *be a feasible integer flow in* G'. *Then there exists a vector* $\boldsymbol{\lambda} = \chi^Q$ *such that* $\boldsymbol{x} = \boldsymbol{x}(\boldsymbol{\lambda})$ *and* Q *is a solution to the VGL problem, where* $\boldsymbol{x}(\cdot)$ *is defined as in Proposition 7.4.*

Proof. The proof is by constructing Q. For each depot $v \in V$ we do the following. For each arc $a \in \delta^-(v)$, we assign x_a of the trucks in $W(v)$ to a. We do this in such a way that each truck is assigned to only one arc. Because $\boldsymbol{x}$ is feasible we have $x_{(v,t)} \leq |W(v)|$, and by flow conservation we have $\boldsymbol{x}(\delta^-(v)) = \boldsymbol{x}(\delta^+(v)) = x_{(v,t)} \leq |W(v)|$, so such an assignment exists. Focus on arc $((C, \boldsymbol{\ell}), v) \in \delta^-(v)$, and let $\{w_1, w_2, \ldots, w_{x_{((C,\boldsymbol{\ell}),v)}}\}$ be the trucks assigned to it. We add the trips $\{(w_k, C, \boldsymbol{\ell}) \mid k = 1, 2, \ldots, x_{((C,\boldsymbol{\ell}),v)}\}$ to Q. We build Q by repeating this for every arc $(u, v) \in A'$ with $u \in \mathcal{S}, v \in V$. Since each truck was assigned to only one arc, Q is indeed a solution and by construction we have that $\boldsymbol{x}(\boldsymbol{\lambda}) = \boldsymbol{x}$. □

Proposition 7.4 and 7.5 allow us to represent a solution to the VGL problem as a flow $\boldsymbol{x}$ in an assignment graph. Manipulating a flow can be done by

maintaining its residual graph, which is well understood (see for example Ahuja, Magnanti and Orlin [5]). In the end, we are only interested in solutions that also satisfy constraints (7.1b). For this reason, we also maintain the *slack* of $\boldsymbol{\lambda}(\boldsymbol{x})$, which is given by

$$\boldsymbol{s}(\boldsymbol{x}) = \boldsymbol{d} - \sum_{r \in \operatorname{supp}(\boldsymbol{\lambda}(\boldsymbol{x}))} \boldsymbol{\ell}^r = \boldsymbol{d} - \sum_{(C,\boldsymbol{\ell}) \in \mathcal{S}} \boldsymbol{\ell}\boldsymbol{x}(\delta^+(C,\boldsymbol{\ell})).$$

The propositions imply that a feasible flow $\boldsymbol{x}$ with $\boldsymbol{s}(\boldsymbol{x}) \geq 0$ can be mapped to a feasible solution to the VGL problem, and vice versa.

7.2.2 LP Rounding

In this section we present a heuristic for constructing a feasible solution for the VGL problem starting from a solution to the LP relaxation of a restricted version of model (7.1). We will refer to this algorithm as the *basic LP rounding algorithm*. How to obtain such an LP relaxation will be explained in detail in Sections 7.3.1–7.3.2.

For now, suppose we have a set of trips $\bar{R} \subseteq R$, and suppose we have an optimal solution $\boldsymbol{\lambda}^*$ to the LP relaxation of model (7.1) with R restricted to $\bar{R}$. Let $\mathcal{S} = \{(C,\boldsymbol{\ell}) \mid (w,C,\boldsymbol{\ell}) \in \bar{R}\}$, and construct the assignment graph $G' = (V', A')$. Order the trips in $\bar{R} = \{r_1, r_2, \ldots, r_{|\bar{R}|}\}$ in non-increasing order of $\boldsymbol{c}(A(C^{r_i}))\lambda^*_{r_i}$.

The idea behind the basic LP rounding algorithm is that we do not want to use trips that are not in the LP solution, we prefer trips that are relatively inexpensive, but we do want to find a feasible integer solution in the end. To achieve this the algorithm maintains a feasible flow $\boldsymbol{x}$ in G' and works in at most $|\bar{R}|$ rounds, starting with round 1. We initialise the flow $\boldsymbol{x} \in \mathbb{R}_+^{A'}$ to $\boldsymbol{x} = \boldsymbol{0}$. In round i, we check whether $\operatorname{supp}(\boldsymbol{s}(\boldsymbol{x})^-) \cap \operatorname{supp}(\boldsymbol{\ell}^{r_i}) \neq \varnothing$. If so, we check whether there exists an augmenting path in the residual graph $G'(\boldsymbol{x})$ from $(C^{r_i}, \boldsymbol{\ell}^{r_i})$ to t. If there exists such a path, we augment $\boldsymbol{x}$ over this path. This completes round i. The algorithm terminates after $|\bar{R}|$ rounds or if $\boldsymbol{s}(\boldsymbol{x}) \geq 0$. If, upon termination, $\operatorname{supp}(\boldsymbol{s}(\boldsymbol{x})^-) = \varnothing$, then the vector $\boldsymbol{\lambda} = \boldsymbol{\lambda}(\boldsymbol{x})$ as in the proof of Proposition 7.5 is a feasible solution to model (7.1). Otherwise, the algorithm has failed.

Ordering the trips in $\bar{R}$ can be done in $O(|\bar{R}| \log |\bar{R}|)$ time. An augmenting path in the residual graph of G' can be found in $O(|A'|)$ time. So, the basic LP rounding algorithm can be implemented to run in $O(|\bar{R}|(|A'| + \log |\bar{R}|))$ time.

The algorithm is a true rounding algorithm in the sense that if you give it a feasible integer solution as input, then it produces a feasible integer solution with the same cost and basically the same structure as output. We end this section by observing that in the light of Theorem 7.3 it is hardly surprising that the LP rounding algorithm may fail to produce a feasible solution. In Section 7.3 we present an algorithm that never fails to find a feasible solution if it exists and one is prepared to wait for it.

7.2.3 The (m, n)-Neighbourhood

Suppose we have a feasible solution for some VGL problem instance that is probably not optimal, for example one that is produced by the basic LP rounding algorithm from Section 7.2.2. We can apply local search starting from this solution to try to find a better one. For this purpose, we introduce the (m, n)-neighbourhood for the VGL problem.

Let $\mathcal{S}$ be the set of all feasible merchant subtours, and let $G' = (V', A')$ be its assignment graph as before. Denote the set of all feasible flows in G' by $F(G')$.

Definition 7.7. Let $\boldsymbol{x} \in F(G')$. A flow $\boldsymbol{x}' \in F(G')$ is an (m, n)-*neighbour* of $\boldsymbol{x}$ if $\boldsymbol{s}(\boldsymbol{x}') \geq \boldsymbol{0}$ and $\boldsymbol{x}' = \boldsymbol{x} - \Delta\boldsymbol{x}^- + \Delta\boldsymbol{x}^+$ for some flows $\Delta\boldsymbol{x}^+, \Delta\boldsymbol{x}^- \in F(G')$ satisfying $\Delta\boldsymbol{x}^- \leq \boldsymbol{x}$, $\Delta\boldsymbol{x}^-(\delta^+(\mathcal{S})) = m$, and $\Delta\boldsymbol{x}^+(\delta^+(\mathcal{S})) = n$. The (m, n)-*neighbourhood* of $\boldsymbol{x}$, denoted by $\mathcal{N}_{m,n}(\boldsymbol{x})$, is the set of all (m, n)-neighbours of $\boldsymbol{x}$.

The (m, n)-neighbourhood consists of all feasible integer solutions that can be obtained from $\boldsymbol{x}$ by cancelling m units of flow, and augmenting n units of flow. In Section 7.2.4 we give algorithms that optimise over $\mathcal{N}_{m,n}$ for certain values of m, n. Next we give a heuristic that can explore $\mathcal{N}_{m,n}$ for any value of m, n in Section 7.2.5.

We overload our (m, n)-neighbourhood terminology by also applying it directly to feasible solutions of the VGL problem:

Definition 7.8. Let $\boldsymbol{\lambda} = \chi^Q$ for some feasible solution Q to the VGL problem. We say that $\boldsymbol{\lambda}' = \chi^{Q'}$, where Q' is a feasible solution to the VGL problem, is an (m, n)-*neighbour* of $\boldsymbol{\lambda}$ if $\boldsymbol{x}(\boldsymbol{\lambda}') \in \mathcal{N}_{m,n}(\boldsymbol{x}(\boldsymbol{\lambda}))$. The (m, n)-*neighbourhood* of $\boldsymbol{\lambda}$, denoted by $\mathcal{N}_{m,n}(\boldsymbol{\lambda})$, is the set of all (m, n)-neighbours of $\boldsymbol{\lambda}$.

Note that if $\boldsymbol{\lambda}'$ is an (m, n)-neighbour of $\boldsymbol{\lambda}$, then the feasible solution $\operatorname{supp}(\boldsymbol{\lambda}')$ need not be a normalised feasible solution. This does not present problems for our heuristics.

7.2.4 Optimising over $\mathcal{N}_{m,n}$

Because the size of the set $\mathcal{S}$ is exponential in $|V|$, we cannot hope to represent the complete assignment graph. The algorithms in this section get around this problem by restricting the graph to the assignment graph made only of the merchant subtours in our current solution to the VGL problem. Given a solution $\boldsymbol{\lambda}$ to model (7.1), we set $\bar{\mathcal{S}} = \{(C, \boldsymbol{\ell}) \mid (r, C, \boldsymbol{\ell}) \in \operatorname{supp}(\boldsymbol{\lambda})\}$.

The $(0, 0)$-neighbourhood of $\boldsymbol{x}$ consists of $\boldsymbol{x}$ only and is of little interest. The $(1, 0)$-neighbourhood consists of all feasible flows $\boldsymbol{x}' \leq \boldsymbol{x}$ with $\boldsymbol{s}(\boldsymbol{x}') \geq 0$ that can be obtained from $\boldsymbol{x}$ by cancelling one unit of flow along a path from a node $u \in \bar{\mathcal{S}}$ to t, and is of interest. Solutions produced by the basic LP rounding algorithm from Section 7.2.2 are not guaranteed to be $(1, 0)$-optimal and can sometimes be improved by a local search algorithm that uses the $(1, 0)$-neighbourhood. Finding a $(1, 0)$-optimal solution is computationally cheap; it

```
neighbour(R̄, λ, m, n)              // searches N_{m,n}(λ)
{
    S := {(C, ℓ) | (w, C, ℓ) ∈ R̄};
    construct the assignment graph G' as in Definition 7.5;
    (λ*, ·) := neighbour(G', x(λ), m, n, |S|);
    return λ*;
}
```

Algorithm 7.3: Exploring $\mathcal{N}_{m,n}(\boldsymbol{\lambda})$: Part I

can be done in $O(|\text{supp}(\boldsymbol{\lambda}(x))|(|K| + \log|\text{supp}(\boldsymbol{\lambda}(x))|)$ time by just sorting the trips $\text{supp}(\boldsymbol{\lambda}(x))$ in non-increasing order of their cost, and processing them in this order by cancelling one unit of flow in $\boldsymbol{x}$ from $(C^r, \boldsymbol{\ell}^r)$ if $\boldsymbol{\ell}^r \leq \boldsymbol{s}(\boldsymbol{x})^+$.

We now consider optimising over the $(m, 1)$-neighbourhood for any fixed $m > 0$. Suppose we have a flow $\boldsymbol{x}' \in F(G')$, and $\text{supp}(\boldsymbol{s}(\boldsymbol{x}')^-) \neq \varnothing$. Our goal is to find a merchant subtour $(C^*, \boldsymbol{\ell}^*)$ that minimises $\boldsymbol{c}(A(C^*))$ such that $\boldsymbol{\ell}^* \geq \boldsymbol{s}(\boldsymbol{x}')^-$, and such that we can augment $\boldsymbol{x}'$ on a path from $(C^*, \boldsymbol{\ell}^*)$ to t (after adding the node $(C^*, \boldsymbol{\ell}^*)$ to our restricted assignment graph). We can restrict ourselves to the case that $\boldsymbol{\ell}^* = \boldsymbol{s}(\boldsymbol{x}')^-$. If such a merchant subtour exists, then it is the cheapest merchant subtour starting from some depot $v \in v(W)$. We can find such a merchant subtour by solving one merchant subtour problem with demand $\boldsymbol{s}(\boldsymbol{x})^-$ and prizes chosen as in Lemma 6.6 for each depot $v \in v(W)$, adding its solution to G' if it covers all demand, searching for an augmenting path from the newly added node in G' to t, and keeping track of the minimum cost merchant subtour for which there exists such an augmenting path. Optimising over the $(m, 1)$-neighbourhood can then be done by trying all $\binom{|\text{supp}(\boldsymbol{\lambda}(\boldsymbol{x}))|}{m}$ possible ways to cancel m units of flow from $\boldsymbol{x}$ to obtain $\boldsymbol{x}'$, and then applying the above procedure to find an $(m, 1)$-neighbour.

Note that in a local search algorithm, optimising over a neighbourhood is an elementary operation that is performed frequently. Therefore optimising over a neighbourhood should be fast in practice if we want our local search to terminate fast, which is what we want if we incorporate local search in a branch-and-bound framework. In this context, solving a lot of $\mathcal{NP}$-hard problems to optimality just to find one improving neighbour will most likely be time-consuming. We therefore do not optimise over the (m, n)-neighbourhood for $(m, n) > (1, 0)$ in our implementations.

7.2.5 A Heuristic for Exploring $\mathcal{N}_{m,n}$

Next we present a computationally cheap heuristic algorithm for exploring $\mathcal{N}_{m,n}$. For this, we assume we have at our disposal a set $\bar{R} \subseteq R$ of trips. Like in the basic LP rounding algorithm, let $\mathcal{S} = \{(C, \boldsymbol{\ell}) \mid (w, C, \boldsymbol{\ell}) \in \bar{R}\}$ be the set of merchant subtours used in $\bar{R}$, and let $G' = (V', A')$ be the assignment graph on

```
neighbour(G', x, m, n, j)                  // searches N_{m,n}(x)
{                                          // returns (λ*, z*), λ* ∈ {0,1}^R, cost z*
    λ* := ∞; z* := ∞;                      // denote S by {u_1, ..., u_{|S|}};
    if (m > 0)                             // cancellation phase
        for (i := 1; i ≤ j; ++i)
            if (x(δ+(u_i)) > 0) {
                cancel one unit of flow in x on a path (u_i ⇝ t);
                if (m > 1) (λ', z') := neighbour(G', x, m − 1, n, i);
                else       (λ', z') := neighbour(G', x, 0, n, |S|);
                if (z' < z*) { λ* := λ'; z* := z'; }
                augment x by one unit on aug. path (u_i ⇝ t) in G'(x);
            }
    else if (n > 0)                        // augmentation phase
        for (i := 1; i ≤ j; ++i) {
            let (·, ℓ) be the merchant subtour u_i;
            if (supp(s(x)^−) ∩ supp(ℓ) ≠ ∅ and
                there exists an augmenting path (u_i ⇝ t) in G'(x)) {
                augment x by one unit on (u_i ⇝ t);
                (λ', z') := neighbour(G', x, 0, n − 1, i);
                if (z' < z*) { λ* := λ'; z* := z'; }
                cancel one unit of flow in x on a path (u_i ⇝ t);
            }
        }
    else if (s(x) ≥ 0) { λ* := λ(x); z* := z_VGL(λ*) }
    return (λ*, z*)
}
```

Algorithm 7.4: Exploring $\mathcal{N}_{m,n}(\boldsymbol{\lambda})$: Part II

$\mathcal{S}$.

Suppose we are given a feasible solution $\boldsymbol{\lambda}$ to the VGL problem. Pseudocode of a recursive algorithm that uses backtracking to enumerate all possible augmentations consisting of cancelling m units and augmenting n units of flow is given in Algorithms 7.3 and 7.4. The function neighbour$(\bar{R}, \boldsymbol{\lambda}, m, n)$ takes as input a set of trips $\bar{R}$, a solution $\boldsymbol{\lambda}$, and parameters m, n, and returns the best (m, n)-neighbour it finds. It constructs the assignment graph and a feasible flow in it from its function arguments and then calls the recursive function neighbour$(G', \boldsymbol{x}, m, n, k)$ that explores the (m, n)-neighbourhood. This is done in two phases, namely one phase in which $m > 0$ where we cancel flow, and one phase in which $m = 0$ and $n > 0$ in which we augment flow. The parameter k is used to eliminate symmetric cancellations and augmentations in the recursion.

At levels $l = 1, \ldots, m$ of the recursion we select nodes $u_{i_l} = (C^l, \boldsymbol{\ell}^l) \in \mathcal{S}$ that are used in the current solution and cancel a trip $(\cdot, C^l, \boldsymbol{\ell}^l)$. After m levels of the recursion, we have cancelled m trips in our solution to the VGL problem, meaning that constraints (7.1b) may no longer be satisfied.

The goal of levels $l = m+1, \ldots, m+n$ of the recursion is to repair this by selecting nodes $u_{i_l} = (C^l, \boldsymbol{\ell}^l)$ and augmenting the flow from this nodes. Hence we can restrict ourselves to augmentations that help in increasing $\boldsymbol{s}(\boldsymbol{x})^-$. Consider level $l \geq m+1$. For all possible merchant subtours $(C^l, \boldsymbol{\ell}^l) \in \mathcal{S}$ that help in reducing the infeasibility of the current solution we try to assign the merchant subtour $(C^l, \boldsymbol{\ell}^l)$ to an available truck, re-arranging the truck assignment if necessary. This is done by augmenting the flow from each node $(C^l, \boldsymbol{\ell}^l)$ in G'. Focus on a particular node $(C^l, \boldsymbol{\ell}^l)$. If we do not succeed in augmenting the flow from $(C^l, \boldsymbol{\ell}^l)$, we know from Menger's theorem (see e.g. Schrijver [105, Theorem 4.1]) that we have saturated some minimum cut separating u_{i_l} and t, which means that there exists a set of trips that we use in the current solution that together use all available trucks at their combined feasible depots. In this case there is no possible way to assign the merchant subtour $(C^l, \boldsymbol{\ell}^l)$ to a trip. Because the order in which we add or remove trips is irrelevant, we demand that for all $k \in \{2, \ldots, m\} \cup \{m+2, \ldots, m+n\}$ we have that $i_{k-1} \geq i_k$.

If, after cancelling m units of flow and augmenting n units of flow we have $\boldsymbol{s}(\boldsymbol{x}) \geq \mathbf{0}$, then we have found a (m,n)-neighbour of $\boldsymbol{\lambda}$. The neighbourhood search returns the best feasible solution to the VGL problem that it encounters.

Searching for an augmenting path in G' can be done in $O(|A'|)$ time, and checking whether $\text{supp}(\boldsymbol{s}(\boldsymbol{x})^-) \cap \text{supp}(\boldsymbol{\ell}) \neq \varnothing$ or $\boldsymbol{s}(\boldsymbol{x}) \geq \mathbf{0}$ can be done in $O(|K|)$ time. Hence, a call of the recursive procedure at any level can be implemented to take $O(|\mathcal{S}|(|A'|+|K|))$ time (at that particular level). It follows that searching $\mathcal{N}_{m,n}(\boldsymbol{\lambda})$ can be done in $O(|\text{supp}(\boldsymbol{\lambda})|^m |\mathcal{S}|^{n+1}(|A'|+|K|))$ time. Because $\text{supp}(\boldsymbol{\lambda}) \subseteq \bar{R}$, $|\mathcal{S}| \leq |\bar{R}|$, and $|A'| \leq |\mathcal{S}||V|$, this is polynomial in the size of $\bar{R}$, V, and K for fixed values of m, n.

7.2.6 The Revised LP Rounding Algorithm

In this section we describe a more careful LP rounding algorithm, that we will refer to as the *revised LP rounding algorithm*. The algorithm maintains a linear program, and works in iterations, starting from 1. The initial linear program is the LP relaxation of model (7.1). Each iteration starts by solving the current LP relaxation. Let $\boldsymbol{\lambda}^i$ be a solution to the LP relaxation of iteration i. For the moment, assume that $\boldsymbol{\lambda}^i$ is optimal. In that case the value of $\boldsymbol{\lambda}^1$ is a lower bound on the value of any feasible integer solution. Round i of the revised LP rounding algorithm works as follows. We first solve the LP relaxation to obtain $\boldsymbol{\lambda}^i$ and try to compute an integer feasible solution $\tilde{\boldsymbol{\lambda}}^i$ starting from $\boldsymbol{\lambda}^i$. We do this by applying the basic LP rounding algorithm from Section 7.2.2 to $\boldsymbol{\lambda}^i$. If the basic LP rounding algorithm algorithm succeeds, we proceed with a local search using the (m,n)-neighbourhood for $(m,n) \in \{(1,0),(1,1)\}$. This gives us $\tilde{\boldsymbol{\lambda}}^i$.

Next, we set up the LP relaxation for iteration $i+1$ as follows. For all trips $r \in R$ with $\lambda_r^i = 1$, and for one trip $r^* = \arg\max_r \{\lambda_r^i \mid \lambda_r^i < 1, r \in R\}$, we add a lower bound $\lambda_r^i \geq 1$ to the LP. Note that $\boldsymbol{\lambda}^i$ is not feasible for this modified LP. Having done this, we proceed with iteration $i+1$.

The algorithm terminates in iteration i if $\boldsymbol{\lambda}^i$ is integer, if the linear program in iteration i is infeasible, or if the value of $\tilde{\boldsymbol{\lambda}}^i$ matches the lower bound obtained in iteration 1. We keep track of the best integer feasible solution that we encounter and report this as the result of the algorithm. Since in each iteration, at least on truck is permanently assigned to a trip, one of these conditions becomes true within $|W|$ iterations.

The algorithm succeeds in finding a feasible solution if at least one of the calls to the basic LP rounding algorithm succeeds. Again, as a consequence of Theorem 7.3 we will have to live with the fact that it might fail.

Note that the algorithm can easily be implemented in an LP-based branch-and-price framework by specifying a unary branching scheme (i.e., one that does not yield a binary branch-and-bound tree, but a "branch-and-bound list", so that each iteration of the rounding algorithm corresponds to an iteration of the branch-and-price algorithm). We also round intermediate feasible solutions to the LP master problem during the column generation algorithm.

In our implementation we use the column generation algorithm as described in Section 3.4.2 to solve the LPs. The pricing problems reduce to the merchant subtour problems (we will come back to this in Section 7.3.2), which we solve using our tabu search heuristic from Section 6.2, with one exception. If we do not find a feasible restricted LP master problem by using the tabu search heuristic to solve the pricing problems in the first iteration, then we use our branch-price-and-cut algorithm for the merchant subtour problem from Section 6.3 to solve the pricing problems to optimality. By doing so we either prove that the instance is infeasible or find an initial feasible restricted LP master problem.

Consider a feasible instance of the VGL problem. In the case that the pricing problems in iteration 1 were solved using the tabu search heuristic, the optimal LP solution $\boldsymbol{\lambda}^1$ to the final restricted LP master problem in iteration 1 need not be optimal to the (unrestricted) LP master problem in iteration 1. Hence, we cannot use its value as a lower bound. To obtain a valid lower bound we run our branch-price-and-cut algorithm for the merchant subtour problem for 32 iterations for each depot. Either we find the optimal solutions to the merchant subtour problems, or we obtain the bounds (2.4) on their value. In both cases we find a bound on the optimal solutions of the pricing problem associated with each available truck. With these bounds we compute the dual bound from Section 3.4.2 on the value of the optimal solution to the VGL instance at hand.

We have implemented the revised LP rounding algorithm. Computational experiments with it are reported on in Section 7.4.

7.3 A Branch-and-Price Algorithm

In this section we describe how to solve the VGL problem using a branch-and-price algorithm. For this purpose we use the branch-and-price algorithm from Section 3.4. The restricted LP formulations, the corresponding pricing problem, and the branching scheme we employ are given in Sections 7.3.1, 7.3.2, and 7.3.3, respectively.

7.3.1 LP Master Problems

We start by describing the relaxations we solve in each node of the branch-and-bound tree and then give an enumeration scheme based on these relaxations. For this purpose, we introduce some extra notation. For the vectors $\boldsymbol{l} = (\boldsymbol{l}_K, \boldsymbol{l}_A)$ and $\boldsymbol{u} = (\boldsymbol{u}_K, \boldsymbol{u}_A)$ with $\boldsymbol{l}_K, \boldsymbol{u}_K \in \mathbb{N}^K$ and $\boldsymbol{l}_A, \boldsymbol{u}_A \in \{0,1\}^A$ and a truck $w \in W$ define the set

$$R_w(\boldsymbol{l}, \boldsymbol{u}) = \{r \in R_w \mid \boldsymbol{l}_A \leq \chi^{(A(C^r))} \leq \boldsymbol{u}_A, \boldsymbol{l}_K \leq \boldsymbol{\ell}^r \leq \boldsymbol{u}_K\}.$$

The set $R_w(\boldsymbol{l}, \boldsymbol{u})$ is the set of feasible trips for truck w that obey the lower and upper bounds $\boldsymbol{l}$ and $\boldsymbol{u}$, respectively. For matrices $L = (L_A L_K), U = (U_A U_K)$ with $L_A, U_A \in \{0,1\}^{W \times A}$ and $L_K, U_K \in \mathbb{N}^{W \times K}$ define

$$R(L, U) = \bigcup_{w \in W} R_w(\boldsymbol{l}_w, \boldsymbol{u}_w).$$

The set $R(L, U)$ is the set of feasible trips obeying for each truck $w \in W$ the lower and upper bounds $\boldsymbol{l}_w$ and $\boldsymbol{u}_w$.

Each node in the branch-and-bound tree will be defined by its own set of bounds. Consider any node in the branch-and-bound tree, and let (L, U) be its bounds. The linear programming problem that we are going to solve in this node is the following:

$$z_{\text{VGL}}(L, U) = \min \quad \sum_{r \in R(L,U)} \boldsymbol{c}(A(C^r))\lambda_r \tag{7.2a}$$

$$\text{subject to} \quad \sum_{w \in W} \sum_{r \in R_w(\boldsymbol{l}_w, \boldsymbol{u}_w)} \boldsymbol{\ell}^r \lambda_r \geq \boldsymbol{d}, \tag{7.2b}$$

$$\boldsymbol{\lambda}(R_w(\boldsymbol{l}_w, \boldsymbol{u}_w)) = 1 \qquad \forall w \in W, \tag{7.2c}$$

$$\boldsymbol{\lambda}_{R(L,U)} \geq \boldsymbol{0}. \tag{7.2d}$$

Associate dual variables $\boldsymbol{\pi}$ and $\boldsymbol{\mu}$ to the constraints (7.2b) and (7.2c), respectively. For $r \in R$, the reduced cost of λ_r is given by $\boldsymbol{c}(A(C^r)) - (\boldsymbol{\ell})^T \boldsymbol{\pi}_{w^r} - \mu_{w^r}$. Because model (7.2) has an exponential number of variables, we solve it by column generation. This involves a restricted model using a subset of the $\boldsymbol{\lambda}$ variables, and yields optimal primal solution $\boldsymbol{\lambda}^*$ to the restricted model together with an optimal solutions $(\boldsymbol{\pi}^*, \boldsymbol{\mu}^*)$ to its dual. If there exists a trip $r = (w^r, C^r, \boldsymbol{\ell}^r)$ such that

$$\boldsymbol{c}(A(C^r)) - (\boldsymbol{\ell})^T \boldsymbol{\pi}^*_{w^r} < \mu^*_{w^r}, \tag{7.3}$$

then we can add λ_r to the restricted model and resolve the resulting linear program. Otherwise, all $\boldsymbol{\lambda}$-variables satisfy their reduced cost optimality conditions, proving that $\boldsymbol{\lambda}^*$ is the optimal solution to model (7.2).

7.3.2 The Pricing Problem

The pricing problem of our branch-and-price algorithm is the problem of finding a trip that satisfies condition (7.3) if one exists. There exists a trip with negative

reduced cost if and only if there exists one for some truck, so we can solve the pricing problem for each truck separately. Consider any iteration of the branch-and-price algorithm, and let (L, U) be its associated bounds. Focus on truck $w \in W$. To solve the pricing problem for truck w, we have to find a trip $(w, C, \boldsymbol{\ell})$ that satisfies the lower and upper bounds $\boldsymbol{l}_w, \boldsymbol{u}_w$ and minimises its reduced cost with respect to dual prices $\boldsymbol{\pi}^*_w$. A merchant subtour $(C, \boldsymbol{\ell})$ satisfies the bounds if $\boldsymbol{l}_w \leq (\chi^{A(C)}, \boldsymbol{\ell}) \leq \boldsymbol{u}_w$. For this purpose, we use the branch-price-and-cut algorithm presented in Chapter 6 with extra bounds on the $\boldsymbol{x}$- and $\boldsymbol{z}$-variables, and objective function $(\boldsymbol{c}_A, \boldsymbol{c}_K) = (\boldsymbol{c}, \boldsymbol{\pi}^*_w)$.

Consider trucks $w, w' \in W$ and suppose that $v(w) = v(w')$, $\boldsymbol{l}_w = \boldsymbol{l}_{w'}$ and $\boldsymbol{u}_w = \boldsymbol{u}_{w'}$. In this case the merchant subtour problems that we have to solve for the pricing problems corresponding to trucks w and w' coincide. In general, the trucks can be divided into classes, each class of trucks having the same pricing problem. We order the trucks in each class, solve the corresponding merchant subtour problem only once to obtain $(C, \boldsymbol{\ell})$, and add a new column only for the first truck w in the order for which condition (7.3) holds.

In the special case that $\boldsymbol{l}_w = \boldsymbol{0}$ and $\boldsymbol{u}_w = \boldsymbol{1}$, we may use the tabu search heuristic from Section 6.2 to solve the pricing problem for truck w. If for some depot $v \in V$ we have that $\boldsymbol{l}_w = \boldsymbol{l}_{w'}$ and $\boldsymbol{u}_w = \boldsymbol{u}_{w'}$ for all $w, w' \in W(v)$ then all trucks in $W(v)$ are in the same class. In this case we have only a single pricing problem for depot v. In the revised LP rounding algorithm of Section 7.2.6 both cases apply simultaneously.

7.3.3 The Branching Scheme

We proceed by giving the branching scheme. We use branching on the original variables as described in Section 3.4.3.

7.4 Computational Results

Ortec Consultants bv kindly supplied us with a 27-city instance of the VGL problem with 236 trucks, stationed non-uniformly at 21 nodes. In this section, we refer to this particular instance as the *master instance* and denote it by the 8-tuple $(G, \boldsymbol{c}, W, K, \boldsymbol{d}, D, \boldsymbol{t}, T)$ where $G = (V, A)$. Our ultimate goal is to design algorithms that can find provably optimal solutions for the master instance, and other instances of the same size. However, at the moment this goal has not yet been achieved. On the contrary, we seem to have quite a lot of work to do before we can even come near.

We tested the revised LP rounding algorithm described in Section 7.2.6 on randomly generated instances that were derived from the master instance as follows. Given a natural number n with $0 < n \leq 27$, we select a set of n nodes $\bar{V} \subseteq V$ out of the $\binom{|V|}{n}$ sets of n nodes uniformly at random. We set $\bar{A} = \bar{V} \times \bar{V}$, $\bar{\boldsymbol{c}} = \boldsymbol{c}_{\bar{A}}$, and $\bar{\boldsymbol{t}} = \boldsymbol{t}_{\bar{A}}$. We use the trucks stationed at $\bar{V}$ by taking $\bar{W} = w(\bar{V})$. We set $\bar{K} = \{(u, v) \in K \mid u, v \in \bar{V}\}$, and $\bar{\boldsymbol{d}} = \boldsymbol{d}_{\bar{K}}$. The resulting instance is $(\bar{G}, \bar{\boldsymbol{c}}, \bar{K}, \bar{W}, \bar{\boldsymbol{d}}, D, \bar{t}, T)$ where $\bar{G} = (\bar{V}, \bar{A})$. Note that there is no guarantee

n	$\lvert W\rvert$	$\lvert K\rvert$	$\lvert v(W)\rvert$	$\boldsymbol{d}(K)$	Fsb/Inf	$\lvert\text{supp}(\boldsymbol{\lambda}^*)\rvert$
4	33.64	9.21	3.54	233.36	24/1	5.42
5	37.21	11.34	3.54	274.77	24/1	8.58
6	46.25	16.91	4.00	412.43	22/3	10.23
7	68.12	31.13	6.11	890.92	22/3	14.00
8	63.40	36.91	6.21	969.84	24/1	19.46
9	76.00	40.68	6.83	1226.88	19/6	23.74
10	86.47	56.34	7.92	1697.32	15/10	29.93
11	94.50	65.19	8.66	1999.24	18/7	35.11
12	106.22	82.44	9.58	2559.10	14/11	41.36
13	110.49	89.15	9.81	2766.97	12/13	47.33

Table 7.1: Statistics of Random VGL Problem Instances

that there are enough trucks in $\bar{W}$ to cover all demand $\bar{\boldsymbol{d}}$ in the available time because of the presence of the constraint on the duration of a trip. In that case the generated instance is infeasible. Even if the generated instance is feasible, our algorithm might not be able to find a feasible solution.

For $n \in \{4, 5, \ldots, 13\}$ we performed 25 runs of our revised LP rounding algorithm. These runs were performed on five 330 MHz Sun Ultra 10 workstations. For all these runs, we were either able to prove that the linear program in the root node was infeasible, or found a feasible solution. In order to give a rough idea of the problem instances, we have gathered some problem statistics in Table 7.1. Here the columns n, $|W|$, $|K|$, $|v(W)|$, $\boldsymbol{d}(K)$, Fsb/Inf, and $|\text{supp}(\boldsymbol{\lambda}^*)|$ denote the number of nodes in G, the average number of available trucks, the average number of commodities, the average number of depots, the average total number of pallets, the number of feasible (Fsb)/infeasible (Inf) instances, and the average number of non-trivial trips in the best known solution taken over all feasible instances, respectively.

The average solution quality, i.e., the average gap between the lower and upper bounds, is depicted in the left part of Figure 7.5. The reported gaps are computed over all feasible instances using the formula $(z^* - \bar{z})/\bar{z} \cdot 100\%$, where z^* is the value of the best known integer solution and $\bar{z}$ the value of the lower bound. The average dimensions of the restricted LP master problems upon termination of the revised rounding algorithm are given in the graph on the right. Running times and run time profiling data are presented in Figure 7.6.

From Table 7.1 we conclude that the probability of generating an infeasible problem instance increases when n increases. Of course, we do not know whether the master instance is feasible. If it is, this would explain the observed behaviour. Otherwise, the probability of generating infeasible instances will decrease again starting from some value of $n \in \{14, \ldots, 27\}$.

On all infeasible instances we solved the problem. On feasible instances the quality of the reported solutions, depicted in the left of Figure 7.5, ranges from within a factor of 1.2 from our best lower bound for instances on four cities to

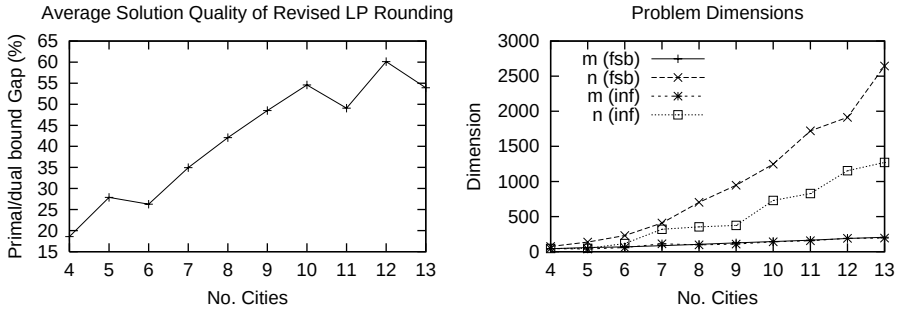


Figure 7.5: Solution Quality of Revised Rounding and LP Dimensions

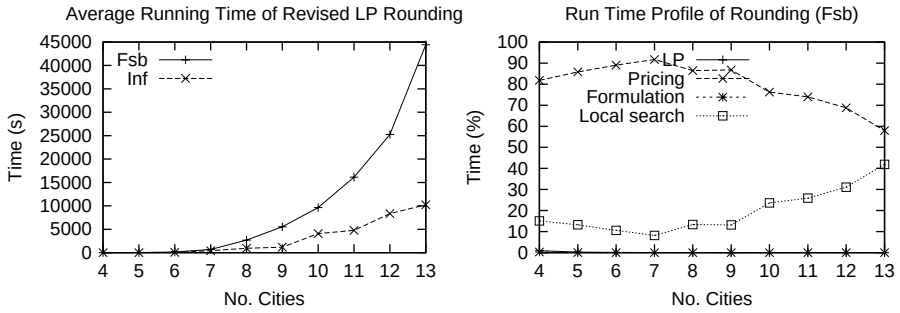


Figure 7.6: Run Time Profiling Data of Revised Rounding

within a factor of 1.65 from our best lower bound for instances on 12 cities. We conclude that the revised rounding algorithm is capable of producing solutions of satisfying quality.

From graph on the right in Figure 7.5 we see that the number of variables in the restricted LP master problem upon termination of the rounding algorithm on feasible instances is twice as large as that on infeasible instances. This suggests that the revised LP rounding algorithm is able to increase the diversity of the trips we use to build the final solution. Also, the number of variables generated stays within 3000.

Next, consider Figure 7.6. The CPU times depicted in the diagram on the left clearly show the main problem of the revised LP rounding algorithm, namely, it is rather slow. This holds especially for the feasible instances. Indeed, it is far too slow to terminate on real-life instances within reasonable time. The CPU time of the revised LP rounding algorithm is divided mainly between two tasks, namely, solving pricing problems and rounding the fractional solutions with the basic LP rounding algorithm and the subsequent local search. In our current implementation we also apply the basic LP rounding algorithm to intermediate solutions obtained within the column generation algorithm. Therefore, it will be possible to decrease the amount of CPU time used for rounding (although

this might be at the cost of the quality of the reported solution). Still, most of the CPU time is consumed by solving pricing problems.

We conclude that our ideal of solving the master problem to optimality is not yet within our reach. Having said this, we do believe that the revised LP rounding algorithm is a good start in the direction of our ideal. Indeed, using this heuristic we are able to find solutions of provable quality to instances that involve the shipping of roughly 2750 pallets with roughly 90 different commodities, using a fleet of roughly 110 trucks.

When comparing the characteristics of our solutions to the characteristics of the solutions that are used in practice by Van Gend & Loos, it turns out that we use a relatively large number of trucks. We believe that this is due to the fact that we do not minimize the number of trucks that are used in non-trivial trips. Note that we can incorporate an upper bound on this number in the models (7.1) and (7.2). This would add a variable to the dual problem, but would not change the structure of the pricing problem (the change in the cost of trivial trips can be handled in the merchant subtour problem by adjusting the cost of using the self-loop at the depot). However, it would force the pricing problems to produce longer trips, that ship more demand. In this way it might allow our heuristics to find solutions that use less trucks.

Bibliography

[1] K.I. Aardal and C.P.M. van Hoesel. Polyhedral techniques in combinatorial optimization II: Applications and computations. *Statistica Neerlandica*, 53:129–178, 1999.

[2] K.I. Aardal, C.P.M. van Hoesel, A. Hipolito, and B. Jansen. Modelling and solving a frequency assignment problem. Unpublished manuscript, 1997.

[3] K.I. Aardal, R. Weismantel, and L.A. Wolsey. Non-standard approaches to integer programming. Technical Report UU-CS-1999-41, Department of Computer Science, Utrecht University, Utrecht, 1999.

[4] E. Aarts and J.K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons Ltd, Chichester, 1997.

[5] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1993.

[6] L. Babel. Finding maximum cliques in arbitrary and special graphs. *Computing*, 46(4):321–341, 1991.

[7] L. Babel and G. Tinhofer. A branch and bound algorithm for the maximum clique problem. *ZOR – Methods and Models of Operations Research*, 34:207–217, 1990.

[8] E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8:146–164, 1975.

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

[10] E. Balas. The prize collecting traveling salesman problem: II. Polyhedral results. *Networks*, 25:199–216, 1995.

[11] E. Balas and M. Fischetti. Lifted cycle inequalities for the asymmetric traveling salesman problem. *Mathematics of Operations Research*, 24(2):273–292, 1999.

[12] E. Balas and J. Xue. Weighted and unweighted maximum clique algorithms with upper bounds from fractional coloring. *Algorithmica*, 15:397–412, 1996.

[13] E. Balas and C.S. Yu. Finding a maximum clique in an arbitrary graph. *SIAM Journal on Computing*, 15:1054–1068, 1986.

[14] C. Barnhart, E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh, and P.H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, 1998.

[15] P. Bauer. The circuit polytope: Facets. *Mathematics of Operations Research*, 22(1):110–145, 1997.

[16] M.S. Bazaraa, J.J. Jarvis, and H.D. Sherali. *Linear Programming and Network Flows*. John Wiley and Sons, New York, 1990.

[17] E.M.L. Beale and J.A. Tomlin. Special facilities in a general mathematical programming system for nonconvex problems using ordered sets of variables. In J. Lawrence, editor, *Proceedings of the Fifth International Conference on Operations Research*, pages 447–454. Tavistock Publications, London, 1970.

[18] M. Bénichou, J.M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1:76–94, 1971.

[19] H.L. Bodlaender. Treewidth: Algorithmic techniques and results. In I. Privara and P. Ruzicka, editors, *Mathematical Foundations of Computer Science 1997, Proceedings of the 22nd International Symposium, MFCS '97*, volume 1295 of *Lecture Notes in Computer Science*, pages 19–36. Springer-Verlag, Berlin, 1997.

[20] R. Borndörfer. *Aspects of Set Packing, Partitioning, and Covering*. PhD thesis, Technische Universität Berlin, Shaker Verlag, Aachen, 1998.

[21] D.P. Bovet and P. Crescenzi. *Introduction to the Theory of Complexity*. Prentice-Hall International (UK) Limited, Hertfordshire, 1994.

[22] A. Caprara and M. Fischetti. $\{0, \frac{1}{2}\}$-Chvátal-Gomory cuts. *Mathematical Programming*, 74:221–235, 1996.

[23] A. Caprara, M. Fischetti, and A. Letchford. On the separation of maximally violated mod-k cuts. In G. Cornuéjols, R.E. Burkard, and G.J. Woeginger, editors, *Integer Programming and Combinatorial Optimization, Proceedings of the 7th International IPCO Conference*, volume 1610 of *Lecture Notes in Computer Science*, pages 87–98. Springer-Verlag, Berlin, 1999.

[24] A. Caprara, M. Fischetti, and A. Letchford. On the separation of maximally violated mod-k cuts. *Mathematical Programming*, 87:37–56, 2000.

[25] G. Carpaneto, M. Dell'Amico, M. Fischetti, and P. Toth. A branch and bound algorithm for the multiple vehicle scheduling problem. *Networks*, 19:531–548, 1989.

[26] R. Carraghan and P.M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9:375–382, 1990.

[27] J. Christensen, J. Marks, and S. Shieber. An empirical study of algorithms for point-feature label placement. *ACM Transactions on Graphics*, 14(3):203–232, 1995.

[28] V. Chvátal. *Linear Programming*. W.H. Freeman and Company, New York, 1983.

[29] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[30] R.G. Cromley. An LP relaxation procedure for annotating point features using interactive graphics. In *AUTO-CARTO 7, Proceedings, Digital Representations of Spatial Knowledge*, pages 127–132, 1985.

[31] H. Crowder, E.L. Johnson, and M.W. Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31:803–834, 1983.

[32] R.J. Dakin. A tree-search algorithm for mixed integer programming problems. *The Computer Journal*, 8:250–255, 1965.

[33] G.B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In T.C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347. John Wiley and Sons, New York, 1951.

[34] G.B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, New Jersey, 1963.

[35] G.B. Dantzig and M.N. Thapa. *Linear Programming, 1: Introduction*. Springer-Verlag, Berlin, 1997.

[36] G.B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8:101–111, 1960.

[37] L. Danzer and B. Grünbaum. Intersection properties of boxes in $\mathbb{R}^d$. *Combinatorica*, 2(3):237–246, 1982.

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

[39] J. Desrosiers, G. Laporte, M. Sauvé, F. Soumis, and S. Taillefer. Vehicle routing with full loads. *Computers & Operations Research*, 15(3):219–226, 1988.

[40] J. Desrosiers, F. Soumis, and M. Desrochers. Routing with time windows by column generation. *Networks*, 14:545–565, 1984.

[41] S. van Dijk, D. Thierens, and M. de Berg. On the design of genetic algorithms for geographical applications. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '99)*, pages 188–195. Morgan Kaufmann, San Fransisco, California, 1999.

[42] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[43] N.J. Driebeek. An algorithm for the solution of mixed integer programming problems. *Management Science*, 12:576–587, 1966.

[44] M. Dror, G. Laporte, and P. Trudeau. Vehicle routing with split deliveries. *Discrete Applied Mathematics*, 50:239–254, 1994.

[45] L. Fleischer. Building chain and cactus representations of all minimum cuts from Hao-Orlin in the same asymptotic run time. *Journal of Algorithms*, 33:51–72, 1999.

[46] B. de Fluiter. *Algorithms for Graphs of Small Treewidth*. PhD thesis, Department of Computer Science, Utrecht University, Utrecht, 1997.

[47] M. Formann and F. Wagner. A packing problem with applications to lettering of maps. In *Proceedings of the 7th Annual ACM Symposium on Computational Geometry*, pages 281–288, 1991.

[48] C. Friden, A. Hertz, and D. de Werra. An exact algorithm based on tabu search for finding a maximum independent set in graph. *Computers & Operations Research*, 17(5):375–382, 1990.

[49] J.M. Gauthier and G. Ribière. Experiments in mixed-integer linear programming using pseudocosts. *Mathematical Programming*, 12:26–47, 1977.

[50] M. Gendreau, G. Laporte, and R. Séguin. A tabu search heuristic for the vehicle routing problem with stochastic demands and customers. *Operations Research*, 44(3):469–477, 1996.

[51] A.M. Geoffrion. Lagrangean relaxation for integer programming. *Mathematical Programming Study*, 2:82–114, 1974.

[52] A. Ghouila-Houri. Caractérisation des matrices totalement unimodulaires. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences (Paris)*, 254:1192–1194, 1962.

[53] R.E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.

[54] R.E. Gomory and T.C. Hu. Multi-terminal network flows. *SIAM Journal on Applied Mathematics*, 9:551–570, 1961.

[55] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, Berlin, 1988.

[56] M. Grötschel and M.W. Padberg. Polyhedral theory. In Lawler et al. [77], pages 251–305.

[57] Z.G. Gu, G.L. Nemhauser, and M.W.P. Savelsbergh. Lifted cover inequalities for 0-1 integer programs: Computation. *INFORMS Journal on Computing*, 10(4):427–437, 1998.

[58] P.L. Hammer, E.L. Johnson, and U.N. Peled. Facets of regular 0-1 polytopes. *Mathematical Programming*, 8:179–206, 1975.

[59] J. Hao and J.B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174, 1992.

[60] A. Hertz, E. Taillard, and D. de Werra. Tabu search. In Aarts and Lenstra [4], pages 121–136.

[61] O.H. Ibarra and C.E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, 1975.

[62] ILOG, Inc., CPLEX Division, Incline Village, Nevada. *CPLEX, a division of ILOG*, 1998.

[63] H. Imai and T. Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *Journal of Algorithms*, 4:310–323, 1983.

[64] D.S. Johnson and L.A. McGeoch. The traveling salesman problem: a case study. In Aarts and Lenstra [4], pages 215–310.

[65] D.S. Johnson and C.H. Papadimitriou. Computational complexity. In Lawler et al. [77], pages 37–85.

[66] E.L. Johnson, G.L. Nemhauser, and M.W.P. Savelsbergh. Progress in linear programming-based algorithms for integer programming: An exposition. *INFORMS Journal on Computing*, 12(1):2–23, 2000.

[67] M. Jünger, G. Reinelt, and S. Thienel. Practical problem solving with cutting plane algorithms in combinatorial optimization. In W. Cook, L. Lovász, and P. Seymour, editors, *Combinatorial Optimization, Papers from the DIMACS Special Year*, volume 20 of *DIMACS Series in Discrete*

Mathematics and Theoretical Computer Science, pages 111–152. American Mathematical Society, 1995.

[68] K.G. Kakoulis and I.G. Tollis. A unified approach to labeling graphical features. In *Proceedings of the 14th Annual ACM Symposium on Computational Geometry*, pages 347–356, 1998.

[69] L.G. Khachiyan. A polynomial algorithm for linear programming. *Doklady Akademii Nauk SSSR 244*, pages 1093–1096, 1979. (In Russian). English Translation: *Soviet Mathematics Doklady 20*, pages 191–194, 1979.

[70] G.A.P. Kindervater and M.W.P. Savelsbergh. Vehicle routing: handling edge exchanges. In Aarts and Lenstra [4], pages 337–360.

[71] G.W. Klau and P. Mutzel. Optimal labelling of point features in the slider model. In D-Z. Du, P. Eades, and X. Lin, editors, *Computing and Combinatorics, Proceedings of the 6th Annual International Conference (COCOON'2000)*, Lecture Notes in Computer Science, 2000. To appear.

[72] M. van Kreveld, T.W. Strijk, and A. Wolff. Point labeling with sliding labels. *Computational Geometry: Theory and Applications*, 13:21–47, 1999.

[73] L. Kučera, K. Mehlhorn, B. Preis, and E. Schwarzenecker. Exact algorithms for a geometric packing problem. In P. Enjalbert, A. Finkel, and K.W. Wagner, editors, *STACS 93, Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, volume 665 of *Lecture Notes in Computer Science*, pages 317–322. Springer-Verlag, Berlin, 1993.

[74] A.H. Land and A.G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[75] G. Laporte and Y. Nobert. Exact algorithms for the vehicle routing problem. *Annals of Discrete Mathematics*, 31:147–184, 1987.

[76] G. Laporte, Y. Nobert, and S. Taillefer. Solving a family of multi-depot vehicle routing and location-routing problems. *Transportation Science*, 22(3):161–172, 1988.

[77] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, editors. *The Traveling Salesman Problem*. John Wiley & Sons Ltd, Chichester, 1985.

[78] E.L. Lawler and D.E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14:699–719, 1966.

[79] J.T. Linderoth and M.W.P. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11(2), 1999.

[80] C. Mannino and A. Sassano. An exact algorithm for the maximum stable set problem. *Computational Optimisation and Applications*, 3:243–258, 1994.

[81] C. Mannino and A. Sassano. Edge projection and the maximum cardinality stable set problem. In D.S. Johnson and M.A. Trick, editors, *Cliques, Coloring and Satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[82] H. Marchand, A. Martin, R. Weismantel, and L.A. Wolsey. Cutting planes in integer and mixed integer programming, 1999. CORE discussion paper 9953, Centre for Operations Research and Econometrics (CORE), Université catholique de Louvain, Belgie.

[83] K. Mehlhorn and S. Näher. *LEDA, A platform for combinatorial and geometric computing*. Cambridge University Press, Cambridge, 1999.

[84] G.L. Nemhauser, M.W.P Savelsbergh, and G.C. Sigismondi. MINTO: A Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1993.

[85] G.L. Nemhauser and G. Sigismondi. A strong cutting plane/branch-and-bound algorithm for node packing. *Journal of the Operations Research Society*, 43(5):443–457, 1992.

[86] G.L. Nemhauser and L.E. Trotter, Jr. Properties of vertex packing and independence system polyhedra. *Mathematical Programming*, 6:48–61, 1974.

[87] G.L. Nemhauser and L.E. Trotter, Jr. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.

[88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[89] G. Neyer. Map labeling with application to graph labeling. In M. Kaufmann and D. Wagner, editors, *Drawing Graphs: Methods and Models*. Teubner, Stuttgart, 2000. To appear.

[90] M.W. Padberg. On the facial structure of set packing polyhedra. *Mathematical Programming*, 5:199–215, 1973.

[91] M.W. Padberg and G. Rinaldi. Facet identification for the symmetric traveling salesman polytope. *Mathematical Programming*, 47:219–257, 1990.

[92] M.W. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[93] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[94] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and Complexity*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[95] J.C. Picard and M. Queranne. On the structure of all minimum cuts in networks. *Mathematical Programming Study*, 13:8–16, 1980.

[96] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[97] Z. Qin, A. Wolff, Y. Xu, and B. Zhu. New algorithms for two-label point labeling. In M. Paterson, editor, *Algorithms — ESA '00, Proceedings of the 8th Annual European Symposium*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2000. To appear.

[98] C.C. Ribeiro and F. Soumis. A column generation approach to the multiple-depot vehicle scheduling problem. *Operations Research*, 42(1):41–52, 1994.

[99] D.J. Rosenkranz, R.E. Stearns, and P.M. Lewis II. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6:563–581, 1977.

[100] F. Rossi and S. Smriglio. A set packing model for the ground-holding problem in congested networks. Technical Report 165, Università di L'Aquila, Dec 1997.

[101] F. Rossi and S. Smriglio. A branch-and-cut algorithm for the maximum cardinality stable set problem. Technical Report 353, "Centro Vito Volterra"-Università di Roma Tor Vergata, Jan 1999.

[102] M.W.P. Savelsbergh and M. Sol. The general pickup and delivery problem. *Transportation Science*, 29:17–29, 1995.

[103] M.W.P. Savelsbergh and M. Sol. DRIVE: Dynamic Routing of Independent VEhicles. *Operations Research*, 46:474–490, 1998.

[104] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons Ltd, Chichester, 1986.

[105] A. Schrijver. A course in combinatorial optimization. CWI, Amsterdam, The Netherlands, 1997.

[106] E.C. Sewell. A branch and bound algorithm for the stability number of a sparse graph. *INFORMS Journal on Computing*, 10(4):438–447, 1998.

[107] T. Strijk, A.M. Verweij, and K.I. Aardal. Algorithms for maximum independent set applied to map labelling. Technical Report UU-CS-2000-22, Department of Computer Science, Utrecht University, 2000. To appear.

[108] T.W. Strijk, 1999. Private communication.

[109] B. Stroustrup. *The C++ programming language, third edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1997.

[110] S. Thienel. *ABACUS, A Branch-And-CUt System, Version 2.0, User's Guide and Reference Manual.* Institut für Informatik, Universität zu Köln, Köln, 1997.

[111] J.A. Tomlin. An improved branch and bound method for integer programming. *Operations Research*, 19:1070–1075, 1971.

[112] L.E. Trotter, Jr. A class of facet producing graphs for vertex packing polyhedra. *Discrete Mathematics*, 12:373–388, 1975.

[113] T.J. Van Roy and L.A. Wolsey. Solving mixed integer programming problems using automatic reformulation. *Operations Research*, 35(1):45–57, 1987.

[114] F. Vanderbeck. A nested decomposition approach to a 3-stage 2-dimensional cutting stock problem. Technical Report 99015, University of Bordeaux 1, Applied Mathematics Lab., 1999.

[115] F. Vanderbeck and L.A. Wolsey. An exact algorithm for IP column generation. *Operations Research Letters*, 19:151–159, 1996.

[116] R.J. Vanderbei. *Linear Programming: Foundations and Extensions.* Kluwer Academic Publishers, Boston, 1996.

[117] A.M. Verweij and K.I. Aardal. An optimisation algorithm for maximum independent set with applications in map labelling. In J. Nešetřil, editor, *Algorithms — ESA '99, Proceedings of the 7th Annual European Symposium*, volume 1643 of *Lecture Notes in Computer Science*, pages 426–437. Springer-Verlag, Berlin, 1999.

[118] A.M. Verweij, K.I. Aardal, and G. Kant. On an integer multicommodity flow problem from the airplane industry. Technical Report UU-CS-1997-38, Department of Computer Science, Utrecht University, Utrecht, 1997.

[119] F. Wagner and A. Wolff. An efficient and effective approximation algorithm for the map labeling problem. In P. Spirakis, editor, *Algorithms — ESA '95, Proceedings of the Third Annual European Symposium*, volume 979 of *Lecture Notes in Computer Science*, pages 420–433. Springer-Verlag, Berlin, 1995.

[120] R. Weismantel. On the 0/1 knapsack polytope. *Mathematical Programming*, 77:49–68, 1997.

[121] A. Wolff and T.W. Strijk. The map labeling bibliography. `http://www.math-inf.uni-greiswald.de/map-labeling/bibliography`.

[122] L.A. Wolsey. Faces for a linear inequality in 0-1 variables. *Mathematical Programming*, 8:165–178, 1975.

[123] L.A. Wolsey. Facets and strong valid inequalities for integer programs. *Operations Research*, 24:367–372, 1975.

[124] L.A. Wolsey. Valid inequalities for mixed integer programs with generalised and variable upper bound constraints. *Discrete Applied Mathematics*, 25:251–261, 1990.

[125] L.A. Wolsey. *Integer Programming.* John Wiley and Sons, New York, 1998.

[126] J. Xue. *Fast algorithms for the vertex packing problem.* PhD thesis, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, Pensylvania, 1991.

[127] S. Zoraster. Integer programming applied to the map label placement problem. *Cartographica*, 23(3):16–27, 1986.

[128] S. Zoraster. The solution of large 0-1 integer programming problems encountered in automated cartography. *Operations Research*, 38(5):752–759, 1990.

[129] P.J. Zwaneveld, L.G. Kroon, and C.P.M. van Hoesel. Routing trains through a railway station based on a node packing model. Technical Report RM/97/030, Maastricht University FdEWB, Maastricht, 1997. To appear in European Journal on Operational Research.

Index of Notation

Sets, Vectors, Matrices

$\mathbb{N}$	4	Natural numbers
$\mathbb{Z}$	4	Integral numbers
$\mathbb{R}$	4	Real numbers
S		Set
2^S	4	Collection of all subsets of S
S^E	4	Set of $\|E\|$-dimensional vectors with elements from S
$\boldsymbol{x}$	4	Vector
$\boldsymbol{x}^T$		Transpose of $\boldsymbol{x}$
$F \subseteq E$		F is subset of E
$F \subset E$		F is proper subset of E
χ^F	4	Incidence vector of F
$\boldsymbol{x}_F$	4	Vector induced by F
$\boldsymbol{x}(F)$	4	Sum of elements of $\boldsymbol{x}$ over F
$\operatorname{supp}(\boldsymbol{x})$	4	Support of $\boldsymbol{x}$
$\boldsymbol{x}^+$	4	Positive part of $\boldsymbol{x}$
$\boldsymbol{x}^-$	4	Negative part of $\boldsymbol{x}$
$S^{I \times J}$	4	Set of $\|I\| \times \|J\|$ matrices with elements from S
A		Matrix
$A_{I'J'}$	4	Matrix induced in A by row/column index sets I'/J'
$A_{J'}$	5	Matrix induced in A by column index set J'
$a_{I'}$	5	Matrix induced in A by row index set I'
A_j	5	The j^{th} column of A
$\boldsymbol{a}_i$	5	The i^{th} row of A (written as a column vector)

Graphs, Directed Graphs

G	5	Directed or undirected graph
V	5	Node set
E	5	Edge set
A	5	Arc set
e	5	Edge
$\{u, v\}$	5	Edge with endpoints u and v

a	5	Arc
(u, v)	5	Arc with tail u and head v
$\delta(S)$	5	Edges with exactly one endpoint in S
$\delta(v)$	5	Edges incident to node v
$\delta^+(S)$	5	Arcs leaving node set S
$\delta^-(S)$	5	Arcs entering node set S
$\delta^+(v)$	5	Arcs leaving node v
$\delta^-(v)$	5	Arcs entering node v
$E(S)$	5	Edge set with both endpoints in S
$A(S)$	5	Arcs with both endpoints in S
$G[S]$	5	Graph induced in G by node set S
$N(S)$	51	Neighbours of node set S
$N(v)$	51	Neighbours of node v
$N_k(S)$	52	k-Neighbourhood of node set S
$N_k(v)$	52	k-Neighbourhood of node v
$\operatorname{diam}(G)$	52	Diameter of G
W	5	Directed or undirected walk
$V(W)$	5	Nodes on walk
$E(W)$	5	Edges on walk
$A(W)$	5	Arcs on walk

Problems, Relaxations, Branch-and-Bound

P	10	Optimisation problem
(X, z)	10	Instance of optimisation problem
$\bar{P}$	10	Relaxation of optimisation problem
$(\bar{X}, \tilde{z})$	10	Instance of relaxation
(X, z, ξ)	10	Instance of decision problem
X^i	12	Feasible set associated with iteration i
$\bar{X}^i$	13	Feasible set of relaxation associated with iteration i
$\bar{X}^i_b$	13	Similar, after branching
$\mathcal{S}$	12	Set of open problems
T	13	Branch-and-bound tree
v_i	13	Node in T associated with iteration i
$\boldsymbol{c}^{\boldsymbol{\pi}}$	16	Reduced cost
P	19	Polyhedron
P^i	21	Polyhedron associated with iteration i
P^i_b	26	Similar, after branching
$\boldsymbol{l}^i$	21	Lower bound associated with iteration i
$\boldsymbol{u}^i$	21	Upper bound associated with iteration i
LP^i	21	LP associated with iteration i

Maximum Independent Set, Map Labelling

P_E	50	Edge formulation of independent set problem

P_{IS}	50	Independent set polyhedron
$P_{\mathcal{C}}$	51	Clique formulation of independent set problem
$P_{\mathcal{C}}^{\mathrm{GUB}}$	51	GUB formulation of independent set problem
$G_{\mathcal{Q}}$	81	Intersection graph of regions in $\mathcal{Q}$
$V_{\mathcal{Q}}$	81	Node set of intersection graph
$E_{\mathcal{Q}}$	81	Edge set of intersection graph
$\mathcal{Q}(P)$	80	Regions of possible labels in map labelling instance P

Merchant Subtour Problem, Van Gend & Loos Problem

$\boldsymbol{c}$	93	Cost vector
K	93	Set of commodities
$\boldsymbol{d}$	93	Demand vector
D	93	Truck capacity
$\boldsymbol{t}$	93	Driving times
T	93	Total available driving time
$\boldsymbol{\ell}$	94	Characteristic vector of load
$(C, \boldsymbol{\ell})$	94	Merchant subtour with route C and load $\boldsymbol{\ell}$
$\mathcal{P}$	98	Set of source–destination paths of any commodity
$\mathcal{P}_k$	98	Set of source–destination paths of commodity k
$\mathcal{P}^C$	96	Set of internal paths along cycle C
$\mathcal{P}_k^C$	96	Similar, source–destination paths of commodity k
W	120	Set of trucks
w	120	Truck
$v(w)$	120	Depot of truck w
$v(W)$	120	Set of depots of trucks in W
$W(v)$	120	Set of trucks with depot v
$(w^r, C^r, \boldsymbol{\ell}^r)$	120	Trip for truck w^r with route C^r and load $\boldsymbol{\ell}^r$
R	124	Set of trips
R_w	124	Set of trips for truck w

Samenvatting

Wij beschouwen het oplossen van wiskundige optimaliseringsproblemen met behulp van een computer. Voor elk gegeven probleem omvat dit het opstellen van een wiskundig model voor het probleem in kwestie, het ontwerpen van een oplossingsmethode die gebaseerd is op dit model, en het implementeren en testen van de ontworpen oplossingsmethode.

De volgende problemen komen in dit proefschrift aan de orde:

(i) Het plaatsen van zoveel mogelijk plaatsnamen op een kaart onder de voorwaarden dat iedere naam geplaatst wordt bij de plaats waar hij bij hoort en dat namen elkaar niet mogen overlappen. Dit probleem is bekend als het *map labelling* probleem.

(ii) Het bepalen van efficiente routes voor een koopman die met een bestelbus door het land reist en geld verdient door winst te maken met in- en verkoop van goederen. Het probleem is, gegeven de prijzen van de goederen in iedere plaats en de kosten van het rijden met de bestelbus, om een optimale dagtrip langs een deelverzameling van de plaatsen uit te rekenen, samen met de bijbehorende in- en verkoop strategie. Hierbij mag de capaciteit van de bestelbus op geen enkel moment gedurende de trip overschreden worden. Dit probleem heet het *merchant subtour probleem*, en een oplossing voor dit probleem noemen wij een *merchant subtour*.

(iii) Het routeren van een collectie vrachtwagens, die gestationeerd zijn op verschillende depots, en waarin we een verzameling klanten hebben met een vraag naar verschillende soorten goederen. Ieder goed heeft een vaste afzender en een vaste bestemming en moet in een geheeltallige hoeveelheid vervoerd worden. De afzender en bestemming zijn deel van de klantenverzameling. Voor het vervoer mogen goederen in geheeltallige hoeveelheden gesplitst worden. Het probleem is, gegeven de kosten van het rijden, om een zodanige routering van de vrachtwagens te vinden plus een laad- en losschema dat aan de vraag wordt voldaan en de totale kosten zo laag mogelijk zijn. Hierbij mag de capaciteit van de vrachtwagens op geen enkel moment overschreden worden. Omdat dit probleem voorkomt bij Van Gend & Loos bv noemen wij dit *het Van Gend & Loos probleem*. Dit probleem, alsmede de probleemgegevens die de situatie beschrijven zoals die

bij Van Gend & Loos voorkomt, werden verkregen van Ortec Consultants bv, te Gouda.

De hierboven genoemde problemen laten zich modelleren als *geheeltallige lineaire programmeringsproblemen*. In dit proefschrift beschrijven wij goed gedefinieerde rekenmethoden, ook wel *algoritmen* genoemd, om met een computer oplossingen voor de genoemde problemen te vinden. De basis voor onze algoritmen wordt gevormd door de zogenaamde *lineaire programmering* in combinatie met het *branch-and-bound* algoritme. De combinatie van lineaire programmering en branch-and-bound is bekend als *LP-gebaseerde branch-and-bound*. Verdere verfijningen van LP-gebaseerde branch-and-bound zijn bekend als *branch-and-price*, *branch-and-cut* en *branch-price-and-cut*.

In een branch-and-cut algoritme wordt het wiskundige model aangesterkt door zogenaamde *geldige ongelijkheden*. Wij maken in onze berekeningen gebruik van *mod-k cuts*, een klasse van algemene geldige ongelijkheden die bekend is uit de vakliteratuur [22, 23, 24]. Daarnaast maken wij gebruik van meer probleemspecifieke geldige ongelijkheden.

Het map labelling probleem laat zich formuleren in termen van een klassiek optimaliseringsprobleem, namelijk het *maximum independent set* probleem. Voor het maximum independent set probleem hebben wij een branch-and-cut algoritme ontwikkeld, alsmede verschillende LP-gebaseerde heuristieken die werken door fractionele oplossingen af te ronden. Om een sterke formulering van het probleem te hebben maken wij gebruik van verschillende klassen van geldige ongelijkheden die bekend zijn uit de vakliteratuur. Onze experimenten tonen aan dat op middelgrote problemen van probleemklassen, die in de vakliteratuur gebruikt worden om algoritmiek voor independent set problemen te testen, ons algoritme in staat om binnen redelijke tijd optimale oplossingen te vinden.

Interessanter zijn de prestaties van ons branch-and-cut algoritme voor het maximum independent set probleem op independent set problemen die verkregen zijn door het herformuleren van map labelling problemen. Map labelling problemen tot 950 steden kunnen wij binnen redelijke tijd optimaal oplossen. Wij zijn er als eerste in geslaagd om map labelling problemen van deze grootte optimaal op te lossen. Daarnaast tonen wij aan dat onze LP-gebaseerde heuristieken optimale of bijna optimale oplossingen geven voor independent set problemen die verkregen zijn door het herformuleren van map labelling problemen.

Voor het merchant subtour probleem hebben wij een branch-price-and-cut algoritme ontwikkeld, alsmede een zogenaamde *tabu search* heuristiek. Ons model voor het merchant subtour probleem is een uitbreiding van het model voor het zogenaamde *price-collecting travelling salesman* probleem [9, 10]. De correctheid van ons model volgt uit het feit dat een speciaal geval van het model beschreven kan worden met een stelsel ongelijkheden dat een mooie wiskundige eigenschap bezit die bekend is als *totale unimodulariteit*. Wij maken gebruik van verschillende geldige ongelijkheden die afkomstig zijn uit het price-collecting travelling salesman probleem. Ons branch-price-and-cut algoritme is in staat om binnen redelijke tijd problemen tot 22 steden optimaal op te lossen. Op deze

klasse problemen vind onze tabu search heuristiek oplossingen die gemiddeld minder dan 3% in waarde afwijken van de optimale oplossingen.

Voor het Van Gend & Loos probleem ontwikkelen wij een branch-and-price algoritme, en een hiervan afgeleide afrondheuristiek. Het branch-and-price algoritme is gebaseerd op een decompositiemethode die bekend is als *Dantzig-Wolfe decompositie*. Door deze methode toe te passen op het Van Gend & Loos probleem is het probleem te vertalen naar deelproblemen voor ieder depot die te interpreteren zijn als merchant subtour problemen en een hoofdprobleem die de resultaten van de deelproblemen combineert. Bij dit combineren worden uit de door de deelproblemen berekende merchant subtours diegene gekozen die gebruikt gaan worden in de oplossing van het Van Gend & Loos probleem. Wij laten door middel van computationele experimenten zien dat onze afrondheuristiek in staat is om op middelgrote problemen oplossingen te vinden die een gemiddelde waarde hebben binnen 60% van onze beste ondergenzen.

Curriculum Vitae

Personalia

Naam	: Abraham Michiel Verweij
Roepnaam	: Bram
Geboortedatum	: 29 oktober 1971
Geboorteplaats	: 's-Gravenhage

Onderwijs

1984–1989 : Hoger Algemeen Voortgezet Onderwijs aan het Griftland College te Soest. Diploma behaald op 15 juni 1989.

1989–1991 : Voorbereidend Wetenschappelijk Onderwijs aan het Griftland College te Soest. Diploma behaald op 11 juni 1991.

1991–1996 : Studie Informatica aan de Universiteit Utrecht. Doctoraal diploma (cum laude) behaald op 26 augustus 1996.

Werkzaamheden

1996–2000 : Assistent in Opleiding bij het Informatica Instituut van de Universiteit Utrecht.