NOTE TO USERS

Université d'Ottawa • University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

BENOÎT, Geneviève

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

M.C.S.

GRADE - DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Finding Violated ut Constraints for the STSP using a Decomposition Approach

Sylvia Boyd

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

L. Moura .....

F. Fiala .....

.....

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES — SIGNATURE — DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Finding Violated Cut Constraints for the STSP Using a Decomposition Approach

Geneviève Benoit

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the Master of Science degree in Computer Science

School of Information Technology & Engineering
Faculty of Engineering
University of Ottawa

Your file Votre référence
ISBN: 0-494-01415-6
Our file Notre référence
ISBN: 0-494-01415-6

Abstract

The Symmetric Travelling Salesman Problem (STSP) is to find a minimum cost Hamiltonian cycle in the weighted complete graph on n nodes. A well known relaxation of this problem is the Subtour Elimination Problem (SEP) which provides very good lower bounds for the STSP.

In this thesis, we develop a new parallel cutting plane approach for solving the SEP. In this approach, many violated cuts are found at each stage by decomposing the current solution into a set of nicely structured points, and then searching for violations in these points in parallel by using an algorithm which exploits this nice structure. Many ideas presented in this thesis could be adapted to other separation problems in combinatorial optimization. We also discuss several enhancements and heuristics which help speed up our implementation of this new parallel cutting plane framework for solving the SEP. We test our implementation and provide results that we believe demonstrate the usefulness of this new parallel algorithm for finding violated cut constraints at each stage of the cutting plane process.

Acknowledgements

My most sincere appreciation goes to my supervisor, Professor Dr. Sylvia Boyd, for her assistance, stimulating suggestions and encouragement throughout the development of this thesis. I will forever be grateful for the countless hours she had devoted to this research and my progress as a mathematician. Her enthusiasm, perseverance and determination are a true inspiration to me.

I would also like to express a special thanks to my husband Louis and my family for their invaluable support, patience and encouragements throughout my study years.

Financial assistance in the form of teaching assistantships from the School of Information Technology and Engineering and the Department of Mathematics at the University of Ottawa as well as a postgraduate scholarship from the Natural Sciences and Engineering Research Council of Canada are also acknowledged with appreciation.

Contents

Chapter 1

Introduction

Consider the following problem. A travelling salesman wishes to travel to a set of cities, visiting each city exactly once, and return to the city at which he or she started in the cheapest possible way.

The above problem is known as the Travelling Salesman Problem and is among the most widely studied topics in Computer Science. It appears to have been formulated in the 1930s [M1930]. Applications of the Travelling Salesman Problem includes printed circuit board production, vehicle routing and job sequencing. Despite the efforts of many, no polynomial-time algorithm is known for solving this problem. In fact, after the theory of NP-completeness was developed, the Travelling Salesman Problem was one of the first problems to be shown to belong to the class of NP-complete problems [K1972]. It is thus considered highly unlikely that a polynomial-time algorithm will ever be found for this problem.

Mathematically, the *Symmetric Travelling Salesman Problem* (henceforth denoted by STSP) is defined as follows: given a graph $G = (V, E)$ and a vector $c \in \mathbb{R}^E$ of edge costs, find a Hamiltonian cycle in G of minimum edge cost. As is often the case, we restrict ourselves to problems in which G is the complete graph (denoted by $K_{|V|}$) since any edge not existing in G can be included with a sufficiently large cost without affecting the optimum solution.

Hence, if we let $K_n = (V, E)$ denote the complete graph on n nodes, then given a

vector $c \in \mathbb{R}^E$ of edge weights, the STSP is to find a tour of K_n of minimum weight, i.e.,

$$min\{cx | x \text{ is a tour of } K_n\}$$

where a *tour* of K_n is the 0-1 incidence vector of the edge-set of a Hamiltonian cycle in K_n. That is, $x \in \{0,1\}^E$ represents a tour of K_n if the set of edges for which $x_e = 1$ form a Hamiltonian cycle in K_n.

When the cost vector satisfies the triangle inequalities, i.e., when $c_{ij} + c_{jk} \geq c_{ik}$ for all $i, j, k \in V$, we call the problem the *metric STSP*.

Given a set S of vectors, the *convex hull* of S, denoted by $Conv(S)$, is the set of all vectors w that can be written as a convex combination of vectors in S. That is, it is the set of all vectors w such that there exists non-negative scalars $\lambda_1, \lambda_2, \ldots, \lambda_k$ and vectors $v_1, v_2, \ldots, v_k$ in S such that $\sum_{i=1}^k \lambda_i = 1$ and $\sum_{i=1}^k \lambda_i v_i = w$. By taking the convex hull of all tours of K_n we obtain the *travelling salesman polytope*, denoted by Q_T^n.

To solve the STSP, we consider the problem

$$min\{cx | x \in Q_T^n\}.$$

Every tour of K_n can be characterized as a spanning subgraph which is both 2-regular (every node has exactly two incident edges) and connected (the subgraph is a single tour, rather than a collection of tours). To translate this into linear constraints on the vector $x \in \mathbb{R}^E$, we first introduce some notation. For any edge-set $F \subseteq E$ and $x \in \mathbb{R}^E$, let $x(F)$ denote the sum $\sum_{e \in F} x_e$. For any node set $W \subset V$, let $\delta(W)$ denote $\{uv \in E | u \in W, v \in V \setminus W\}$. We write $\delta(v)$ instead of $\delta(\{v\})$ for $v \in V$. The sets W and $V \setminus W$ are called the *shores* of the *cut* $\delta(W)$.

The 2-regular requirement implies that any $x \in Q_T^n$ must satisfy

$$x(\delta(v)) = 2 \text{ for all } v \in V. \tag{1.1}$$

These n equations are called the *degree constraints* for Q_T^n.

Moreover, given any vector $x \in Q_T^n$, the connectivity requirement can be translated as

$$x(\delta(S)) \geq 2 \text{ for all } S \subseteq V, 2 \leq |S| \leq n-2. \tag{1.2}$$

These constraints are called the *subtour elimination constraints*, or the *cut constraints* and were first introduced in [DFJ1954]. These constraints say that for any set $S \subseteq V, 2 \leq |S| \leq n-2$, a tour must enter and leave S at least once, and therefore will contain at least two edges from $\delta(S)$. Hence these constraints force the support graph of x, i.e., the subgraph of K_n induced by the edges in x which have $0 < x_e \leq 1$, to be connected. It is clear that we only need to consider the inequalities in (1.2) for which $|S| \leq \lfloor n/2 \rfloor$ since S and $V \setminus S$ result in the same inequality.

Note that if the degree constraints are satisfied, then by taking $|S| = 2$ in the inequalities (1.2), we obtain the so-called *upper-bound constraints*

$$x_e \leq 1 \text{ for all } e \in E. \tag{1.3}$$

Similarly, the upper-bound constraints and the degree constraints will force the constraints $x(\delta(S)) \geq 2$ to be satisfied for every subset S of nodes with $|S| = 2$.

In 1954, Dantzig, Fulkerson and Johnson have shown that the STSP can be formulated as the following integer linear program ([DFJ1954]):

$$\begin{aligned}
&\text{minimize } cx && && (1.4)\\
&\text{subject to: } x(\delta(v)) = 2 && \text{for all } v \in V, && (1.5)\\
&x(\delta(S)) \geq 2 && \text{for all } S \in \mathcal{S}, && (1.6)\\
&0 \leq x_e \leq 1 && \text{for all } e \in E, && (1.7)\\
&x && \text{integer.} && (1.8)
\end{aligned}$$

where $\mathcal{S} = \{S \subset V, 3 \leq |S| \leq \lfloor n/2 \rfloor\}$.

The STSP is known to be an NP-hard problem, even in the metric case [JP1985]. One approach taken for finding reasonably good solutions is to look for a k-approximation

algorithm for the problem, i.e., try to find a heuristic for the STSP which finds a tour which is guaranteed to be of cost at most k times the cost of the optimal tour for some constant $k \geq 1$. Currently the best k-approximation algorithm known for the metric STSP is Christofides algorithm [C1976] for which $k = 3/2$. Surprisingly, no one has been able to improve upon this algorithm in over two decades. Note that for general costs there does not exist a k-approximation algorithm unless $P = NP$ [JP1985].

One direction which seems promising for finding improved solutions for the STSP is the study of a linear relaxation of this problem called the *Subtour Elimination Problem* (henceforth denoted by SEP). This relaxation can be obtained from the integer linear programming formulation for the STSP by dropping the integer requirement (1.8), i.e., it is the linear program:

$$\begin{aligned} & \text{minimize } cx & & \\ \text{subject to: } & x(\delta(v)) = 2 & & \text{for all } v \in V, \\ & x(\delta(S)) \geq 2 & & \text{for all } S \in \mathcal{S}, \\ & 0 \leq x_e \leq 1 & & \text{for all } e \in E. \end{aligned} \tag{1.9}$$

The associated *subtour polytope*, which we denote by Q_S^n, is defined by

$$Q_S^n = \{x \in \mathbb{R}^E | x \text{ satisfies } (1.5), (1.6), (1.7)\}. \tag{1.10}$$

Since SEP is a relaxation of STSP, we have $Q_T^n \subseteq Q_S^n$. Hence $\min\{cx | x \in Q_S^n\} \leq \min\{cx | x \in Q_T^n\}$. To measure the quality of the lower bound provided by the SEP for the STSP, people have been studying the *integrality gap* α for the SEP, which is the worst-case ratio between the optimal solution for STSP and the optimal solution for SEP, i.e.

$$\alpha = \max_{c \geq 0} \left(\frac{\{cx | x \in Q_T^n\}}{\{cx | x \in Q_S^n\}} \right).$$

It is clear that the value α gives one measure of the quality of the lower bound provided by the SEP for the STSP. Moreover, a constructive proof for value of α would provide an α-approximation algorithm for the STSP.

It is known that for the metric STSP, α is at most 3/2 ([SW1990], [W1980]), however, no example for which this ratio comes close to 3/2 has yet been found. In fact, a well-known conjecture states the following (see [BL2002] for more details):

Conjecture 1.0.1 *For the metric STSP, the integrality gap for the SEP is* 4/3.

Thus solving the SEP provides a very good lower bound on the optimal value for the STSP.

Moreover, because of the quality of the lower bound provided by the SEP solution, the optimal SEP solution would provide a very good starting point for the cutting plane approach for solving the STSP (for more information on the cutting plane method, see the paragraphs which follow).

From the above discussion, it is clear that efficient methods for solving the SEP are very sought after in order to obtain improved STSP solutions.

Note however that plugging the linear program (1.9) into a commercial LP solver is not trivial since there exists an exponential number of subtour constraints. Nevertheless, in most cases it is still possible to solve the SEP efficiently by using a cutting plane approach.

George Dantzig, Ray Fulkerson and Selmer Johnson first introduced the cutting plane method for solving the TSP in 1954 (see [DFJ1954] for more details). They illustrated the power of this method by solving an instance with 49 cities to optimality, an impressive size at that time. More recently, the method has been used to solve much larger practical instances of the TSP to optimality (see [PR1991], [JRR1995] and [ABCC1995], for example). Moreover, since then, the cutting plane method has been generalized and successfully applied to many other problems in combinatorial optimization. In the next section, we discuss the cutting plane approach for solving

the SEP. Note that the cutting plane algorithm is one of the most successful methods known for solving the SEP.

1.0.1 Cutting plane algorithm for solving the SEP

In this approach, a commercial linear programming code is used to optimize the objective function over a small subset of the constraints defining Q_S^n. If the optimal solution obtained, denoted by x^*, corresponds to a point of Q_S^n, then x^* also corresponds to an optimal solution to the SEP. Otherwise, we look for a valid constraint for Q_S^n which is violated by x^*, add this constraint to our present subset of constraints, and repeat.

The system of constraints most often used as a starting subset for the cutting plane approach for the SEP is the following:

$$\begin{aligned} &x(\delta(v)) = 2 \text{ for all } v \in V \\ &0 \leq x_e \leq 1 \text{ for all } e \in E. \end{aligned} \tag{1.11}$$

The polytope associated with the above set of constraints is called the *fractional 2-factor polytope* and it is denoted by Q_F^n.

The core of the cutting plane algorithm is identifying a violated constraint for the current optimal solution. This problem is called the *separation problem*. More formally, given any point $x \in \mathbb{R}^E$ and a family $\mathcal{F}$ of inequalities in $\mathbb{R}^E$, the separation problem consists of identifying one or more inequalities violated by x or proving that no such inequalities exist.

Given a family of inequalities $\mathcal{F}$, we call a procedure that solves the separation problem for $\mathcal{F}$ *exact* and we call a procedure that sometimes identifies violated inequalities, but which does not guarantee the solution to the separation problem *heuristic*.

For the subtour elimination constraints, given any point x in Q_F^n, the separation problem reduces to finding the minimum weight cut in the support graph of x. If

the minimum cut, $\delta(S)$, has value less than two, then $x(\delta(S)) \geq 2$ is a most-violated constraint. Otherwise, we have that Constraint (1.6) is satisfied for all $S \in \mathcal{S}$ and thus, x also belongs to Q_S^n. There are several polynomial-time algorithms for finding minimum cuts in undirected graphs; the Gomory-Hu algorithm, for example, runs in $O(n^2 m \log(n^2/m))$ time on a graph with m edges and n nodes [JRR1995] . Therefore, the separation problem for the subtour elimination constraints can be solved in polynomial-time. Consequently, it is possible to optimize over Q_S^n in polynomial-time by means of the ellipsoid algorithm (see [GLS1981] for more details). However, the times involved in this polynomial-time algorithm are not practical, and a practical polynomial-time algorithm to solve the problem of optimizing over Q_S^n is not known.

Although some very fast heuristic algorithms exist for finding a minimum cut in a graph, these algorithms are only designed to find one violation. Considering the overhead associated with the run of the LP solver, adding only one violated constraint at a time seems very inefficient. Therefore, in practice, people tend to use very fast heuristics to find a larger portion of the violated cut constraints at each step of the cutting plane process. By doing so, they usually achieve a shorter overall running time.

In this thesis, we investigate a new method for finding violated subtour elimination constraints in a cutting plane approach for solving the Subtour Elimination Problem which is particulary well-suited for parallel computers. This method finds a large portion of violated subtour constraints at each step of the cutting plane process using a decomposition approach. The approach presented here could easily be adapted and applied to many other separation problems in combinatorial optimization. It is based on the following two very important results:

Lemma 1.0.1 *Let $z \in Conv(Q)$ for some finite set Q of vectors. If an inequality $ax \geq b$ is satisfied for every vector x in Q, then we also have $az \geq b$.*

Proof Since $z \in Conv(Q)$ we can write $z = \sum_{x \in Q} \lambda_x x$ with $\sum_{x \in Q} \lambda_x = 1$, $\lambda_x \geq 0$ for all $x \in Q$. Hence, $az = a \sum_{x \in Q} \lambda_x x = \sum_{x \in Q} \lambda_x ax \geq \sum_{x \in Q} \lambda_x b = b \sum_{x \in Q} \lambda_x = b$. □

Corollary 1.0.2 *Let $z \in Conv(Q)$ for some finite set Q of vectors. The only inequalities which can be violated by z are the ones which are violated by at least one vector in Q.*

Hence, in order to find the complete list of constraints violated by any point z in $Conv(Q)$ if suffices to find the complete list of violations in each of the vectors $x \in Q$ and verify each one of the violations found in z.

Since finding the complete list of violations in a vector $x \in Q$ can be done independently without considering the other vectors in Q, we can use several processors in parallel to find the complete list of violations in each one of the vectors $x \in Q$. This observation will have a significant impact on the success of this new cutting plane algorithm. Of course, in order to efficiently make use of this idea, it must be easier to find the complete list of violations in the vectors of Q than in the original vector z. Otherwise, we have not simplified anything.

This approach for finding violated constraints for a problem has already been introduced in [RKPT2003]. In that paper, the authors describe how they were able to successfully apply this idea to separate over the capacity constraints of the Capacitated Vehicle Routing Problem by taking advantage of their ability to solve small instances of the TSP efficiently. Using this decomposition-based methodology for finding violated constraints, they were able to solve many large instances of the Vehicle Routing Problem to optimality and some of those for the first time.

1.0.2 Applying this idea to the separation problem associated to the subtour elimination constraints

If System (1.11) is used as the starting set of constraints for the cutting plane approach for the Subtour Elimination Problem, then any point z obtained during the cutting plane process will be a point in the fractional 2-factor polytope Q_F^n. It will therefore

be possible to write z as a convex combination of a finite subset of vertices of Q_F^n (see Section 2.4 for more details). Moreover, Carathéodory's Theorem assures us that at most $O(n^2)$ vertices will be required (see [L1982]). Subsequently, if we let $\overline{Q}$ denote the set of vertices of the polytope Q_F^n, then we can write z as a convex combination of a finite number of vectors in $\overline{Q}$, i.e., we can find a finite subset Q of $\overline{Q}$ such that z can be written as $z = \sum_{x \in Q} \lambda_x x$, where the λ_x's satisfy $\lambda_x > 0$ for all $x \in Q$ and $\sum_{x \in Q} \lambda_x = 1$. Hence, using the result of Corollary 1.0.2, we know that if a subtour constraint is violated by the point z then it must also be violated by at least one vertex x present in the convex combination of z.

For example, consider the point z whose support graph is shown in Figure 1.1 and consider the decomposion for z which is also suggested in Figure 1.1. It is easy to verify that the first two vertices in the decomposition satisfy every subtour elimination constraints for Q_S^{10}. However, the third vertex violates the subtour elimination constraints $x(\delta(S)) \geq 2$ where S corresponds to the node-set which is circled. If we verify the constraint corresponding to this cut in the original point z we obtain a violation in z. It is also very easy to verify that this cut is the only cut constraint violated by z.

As will be mentioned in Chapter 4, the vertices of the fractional 2-factor polytope have a very special structure: any vertex $\hat{x}$ of Q_F^n is half-integer, and the set of edges $J \subseteq E$ such that $\hat{x}_j = \frac{1}{2}$ for all $j \in J$ partitions into the edge-set of an even number of odd disjoint cycles. Hence, it might be possible to exploit the well-defined structure of the vertices of the fractional 2-factor polytope to construct an efficient algorithm for finding the complete set of violated cut constraints in any vertex $\hat{x}$ of Q_F^n. Since any point z in the fractional 2-factor polytope can be written as a convex combination of vertices of Q_F^n, such an algorithm will allow us to use the result of Corollary 1.0.2 to find violated subtour constraints in z.

The main objective of this thesis is to develop a new algorithm for finding violated subtour constraints based on the decomposition methodology described in this chapter

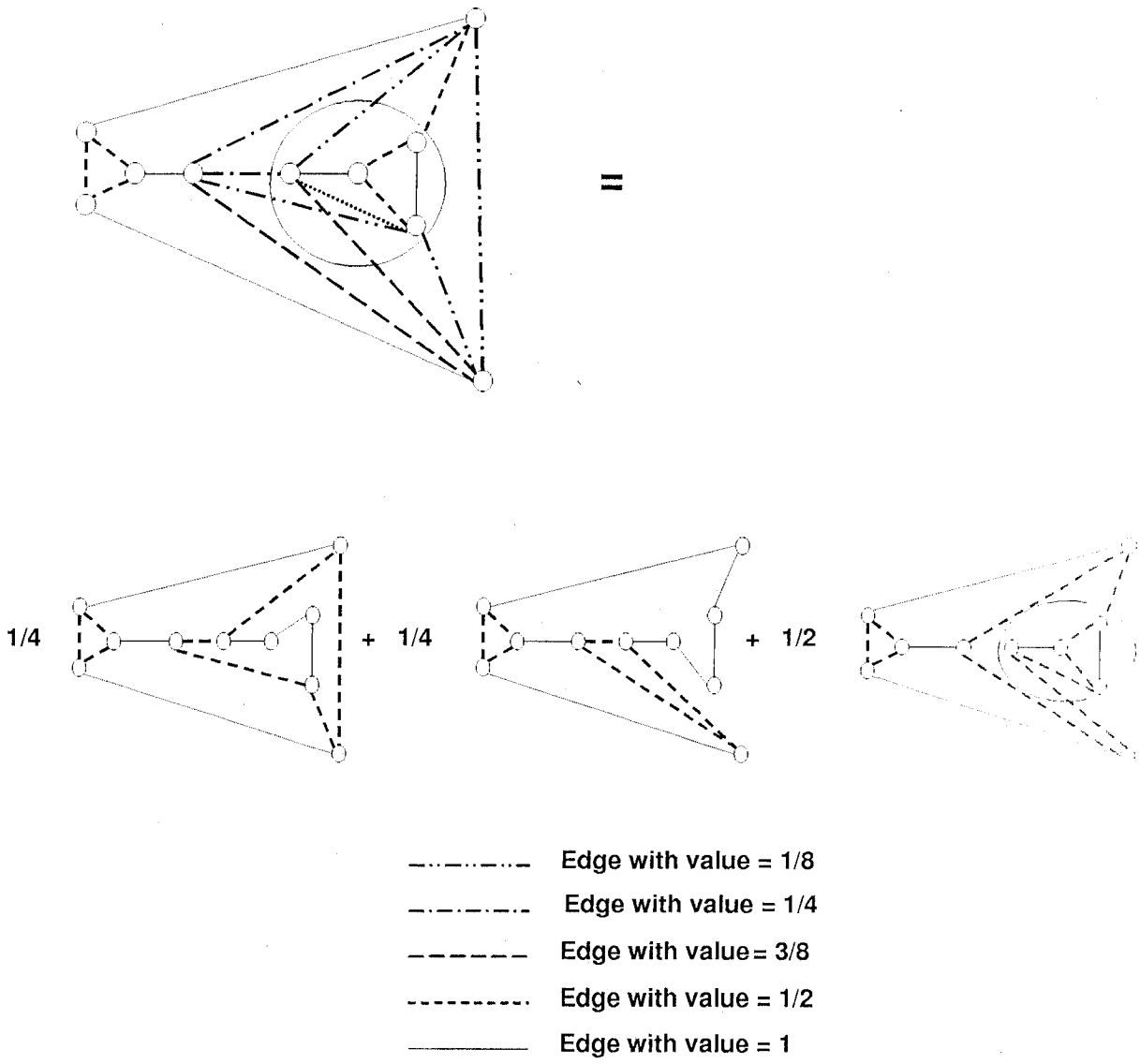

Figure 1.1: Decomposition based separation

and implement this algorithm into a cutting plane framework for solving the Subtour Elimination Problem.

As we will show in this thesis, although this method of finding violated constraints had already been introduced and successfully applied to other problems in combinatorial optimization, we were able to improve the method by developing two new

enhancements which help speed up the decomposition process immensely. Moreover, as we show in Section 3.3, these enhancements can be applied in a straightforward manner to many other problems in combinatorial optimization. To efficiently make use of the ideas presented in this chapter, we had to exploit the well-defined structure of the vertices of the fractional 2-factor polytope in order to develop an efficient algorithm for finding the complete set of violated cut constraints in any vertex of the fractional 2-factor polytope. As we will show in Chapter 4, although a vertex of the fractional 2-factor polytope can have an exponential number of violated constraints, we were able to construct an exact separation algorithm for the subtour elimination constraints. Note that in [RKPT2003], this decomposition-based methodology is only used as a heuristic for finding violated constraints.

In Chapter 2 of this thesis, we introduce the background notation and theory required for this thesis. Note that some of the definitions which will be given have already been stated. However, for the convenience of the reader, we have decided to include them again.

In Chapter 3, we present an efficient algorithm for writing any point in the fractional 2-factor polytope as a convex combination of a finite number of vertices of Q_F^n. The decomposition algorithm presented has previously been introduced in [RKPT2003]. In the same chapter, we describe two new enhancements we have made to the algorithm, one of which greatly reduces the running time of the decomposition algorithm. As the reader will notice, these new enhancements to the decomposition algorithm could be applied in a straightforward manner to other combinatorial optimization problems.

In Chapter 4 of this thesis, we present an efficient algorithm we have constructed to find violated subtour elimination constraints in any vertex of the fractional 2-factor polytope. Recall that if a point z in Q_F^n is written as a convex combination of vertices of Q_F^n, then finding the complete set of violations in z can be achieved by finding the complete set of violations in each one of the vertices present is the decomposition of z

and verifying each one of these constraints in z. However, as we will show in Chapter 4, a vertex of Q_F^n can have an exponential number of violated subtour constraints. Hence, verifying each one of these constraints in z can be impractical. In Chapter 4, we demonstrate how we are able to overcome this obstacle and construct an exact separation algorithm for any point of the fractional 2-factor polytope.

In Chapter 5, we give some details concerning the implementation of this cutting plane algorithm. In particular, we discuss several enhancements we made to speed up the process. Moreover, we explain how the cutting plane framework had to be adapted to solve large-scaled Subtour Elimination Problems.

In Chapter 6, we report our test results. In particular, we discuss the effectiveness of the decomposition algorithm presented in Chapter 3 and the algorithm for finding violated cut constraint in any vertex of Q_F^n described in Chapter 4. We also discuss the computational results obtained from our implementation of this cutting plane algorithm. Moreover, we show how relatively well our algorithm for finding violated subtour constraints performed when compared to the cut generator implemented in CONCORDE. CONCORDE is a very powerful computer code designed specifically to work with the TSP developed by David Applegate, Robert Bixby, Vasek Chvátal and Bill Cook.

Finally, in Chapter 7, we conclude with some closing remarks and ideas for future research.

Chapter 2

Background and Notation

2.1 General Notation

For any finite set E, let $\mathbb{R}^E$ denote the set of all real vectors indexed by E, and let $|E|$ represent the *cardinality* of E. Given another set $F \subseteq E$, we let $E \setminus F$ denote the members of E which are not members of F. For $x \in \mathbb{R}^E$ and $F \subseteq E$, let $x(F) = \sum(x_e : e \in F)$.

Let V be a non-empty set and let $C = \{C_1, C_2, \ldots, C_k\}$ where $C_i \subseteq V$ for $i = 1, 2, \ldots k$. We say that the sets C_i and C_j *cross* if $C_i \cap C_j \neq \emptyset$ and neither $C_i \subseteq C_j$ nor $C_j \subseteq C_i$. We call C a *nested family* if it contains no pairs of sets which cross.

For any two finite sets J and K, we let $\mathbb{R}^{J \times K}$ denote the set of all real matrices whose rows are indexed by J and whose columns are indexed by K. For any $B \subseteq K$, we let A_B denote the matrix whose columns are those of A indexed by B. If $B = \{i\}$, we use A_i rather than $A_{\{i\}}$.

For any positive integer n, the *identity matrix* of size n, denoted by I^n, is the matrix with n rows and n columns such that the diagonal entries have value one and all other entries have value zero. For any matrix $A \in \mathbb{R}^{J \times K}$, the inverse of A, denoted by A^{-1}, is defined by $A^{-1}A = I^{|K|}$ if such a matrix exists.

For any finite sets J and K and matrix $A \in \mathbb{R}^{J \times K}$, the *transpose* of A, denoted by A^t, is the matrix whose columns are the rows of A and whose rows are the columns of A. In this thesis, we write A instead of A^t unless it is not absolutely apparent from the context.

2.2 Graph Theory

A *graph* G is an ordered pair (V, E) where V is a finite set of elements called *nodes*, and E is a finite set of elements called *edges* such that every edge $e \in E$ corresponds to two distinct nodes in V, called the *ends* of e. An edge $e \in E$ with ends u and v is denoted by uv, and we say that e *joins* u and v. Two nodes u and v in V are said to be *adjacent* if $uv \in E$, and if $e = uv$, e is said to be *incident* with u and v.

There exists many ways of representing a graph on a computer. Since at every step of the cutting plane process we only considered a small portion of the edge-set of the complete graph K_n (see Section 5.4), all the graphs we considered in our implementation were sparse. Hence the representation we decided to use is an *adjacency list* data structure. The adjacency list representation of a graph $G = (V, E)$ consists of an array Adj of $|V|$ linked lists, one for each node in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the nodes v such that $uv \in E$. That is, $Adj[u]$ constists of all the nodes adjacent to u in G.

If G is any graph, we let $V(G)$ and $E(G)$ denote the node-set and edge-set of G respectively. For any $S \subseteq V(G)$ we use $\gamma(S)$ to denote the set of edges in $E(G)$ with both ends in S. Moreover for any two subsets of $V(G)$, S and T, we use $\delta(S)$ to denote the set of edges in $E(G)$ with exactly one end in S and we use $E(S : T)$ to denote the set of edges in $E(G)$ with one end in S and one end in T. We write $\delta(v)$ instead of $\delta(\{v\})$ for $v \in V$. For any node $v \in V(G)$, the *degree* of v is $|\delta(v)|$. Moreover, we define a cut in a graph G as the edges $\delta(S)$ for some set $S \subset V$. By removing the set $\delta(S)$ from $E(G)$, we obtain a partition of $V(G)$ into two sets S and $V(G) \setminus S$. The sets

S and $V(G) \setminus S$ are called the shores of the cut $\delta(S)$. Given a weighted graph G with weight vector $x \in \mathbb{R}^{E(G)}$ and a set $S \subset V(G)$, the value of a cut $\delta(S)$ in G is given by $x(\delta(S))$.

A graph G is *complete* if every pair of nodes in G is joined by exactly one edge. The complete graph on n nodes is denoted by K_n. For the remainder of this thesis, we use E and V to denote the edge-set and the node-set of the complete graph K_n, unless otherwise stated.

A graph H is called a *subgraph* of G if $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$ and every edge $e \in E(H)$ has the same ends in H as in G. If $V(H) = V(G)$ then H is a *spanning* subgraph, and if every edge of G in $\gamma(V(H))$ is in $E(H)$ then H is an *induced* subgraph of G.

A *path* in a graph G is a finite non-null sequence $v_0 e_1 v_1 e_2 v_2 \dots e_k v_k$ whose terms are alternately nodes and edges such that the nodes are distinct, and for $1 \leq i \leq k$ the ends of e_i are v_{i-1} and v_i. We say path P *joins* v_0 and v_k, and sometimes we denote P by the node sequence $v_0, v_1, \dots, v_k$, where $V(P) = \{v_0, v_1, \dots, v_k\}$ and $v_{i-1}v_i \in E(P)$ for $i = 1, 2, \dots, k$. The *length* of P is $|E(P)|$, and a *Hamiltonian path* of G is a path P in G of length $|V(G)| - 1$.

A graph G is *connected* if every pair of nodes in G are joined by a path. A *component* of a graph G is any maximal connected subgraph of G.

A *cycle* in G is a connected subgraph C of G such that every node in $V(C)$ has degree 2 in C. Sometimes we denote C by the node sequence $v_0, v_1, \dots, v_k$, where $V(C) = \{v_0, v_1, \dots, v_k\}$, $v_0 v_k \in E(C)$ and $v_{i-1}v_i \in E(C)$ for $i = 1, 2, \dots, k$. The *length* of C is $|E(C)|$, and a *Hamiltonian cycle* of G is a cycle C in G of length $|V(G)|$.

The *node-edge incidence matrix* of a graph G is a matrix $A \in \mathbb{R}^{V(G) \times E(G)}$ such that the entry of A indexed by node v and edge e has value 1 if v is an end of e, otherwise it has value 0. Also, for any vector $x \in \mathbb{R}^{E(G)}$, the *support graph of* x, denoted by G_x, is the graph whose node-set is $V(G)$, and whose edge-set consists of all edges $e \in E(G)$

such that $x_e \neq 0$. Given $x \in \mathbb{R}^{E(G)}$, the *fractional support graph of* x is the graph whose node-set is $V(G)$, and whose edge-set consists of all edges $e \in E(G)$ which have a fractional value.

A *multigraph* is a graph which admits multiple edges between the same nodes. Moreover, given any graph G and a node-set $S \subset V(G)$, we let $G \circ S$ denote the graph obtained from G by shrinking the node-set S, i.e., $G \circ S$ is the graph obtained by replacing the complete set of nodes in S by a single node w and by replacing each edge uv with $u \in S$ and $v \notin S$ by an edge wv. Note that this process might create multiple edges.

In the remainder of this thesis, in any weighted graph drawn, the edges of value $\frac{1}{2}$ are represented by a dashed edge and the edges of value 1 by a solid edge. All other non-zero edges have their value beside them and edges not shown have value 0.

2.3 Linear Programming

A *linear programming problem* involves optimizing a linear objective function subject to a finite number of linear constraints. For our purposes, it is convenient to consider linear programming problems having the following form:

$$\begin{aligned} &\text{minimize } cx \\ &\text{subject to: } Ax = b \\ &\qquad\qquad Dx \geq d \\ &\qquad\qquad x \geq 0. \end{aligned} \tag{2.1}$$

Note that any general linear programming problem can be transformed into the above form (see [D1963]) and all theorems in this section can be extended in the appropriate way.

A vector x is a *feasible solution* to (2.1) if it satisfies all of the given constraints.

A feasible solution is an *optimal solution* for (2.1) if it minimizes cx for all feasible solutions x.

The following is a consequence of the fundamental theorem of linear programming (see [D1963]).

Theorem 2.3.1 *For any linear programming problem exactly one of the following situations occurs:*

i) There exists no feasible solution.

ii) The objective function is unbounded subject to the constraints.

iii) There is an optimal feasible solution.

With any linear programming problem we can associate a *dual* linear programming problem. The primal and dual problems have the property that the value of the objective function at a feasible solution in one bounds the optimal objective value in the other.

The following is the dual linear programming problem associated with (2.1):

$$\begin{aligned} &\text{maximize } yb + wd \\ &\text{subject to: } yA + wD \leq c, \\ &\quad w \geq 0, y \text{ unrestricted.} \end{aligned} \tag{2.2}$$

Note that every constraint in the primal problem corresponds to a variable in the dual problem and vice versa.

The following three theorems describe the relationship between (2.1) and (2.2).

Theorem 2.3.2 (Weak L.P. Duality Theorem) *(see [D1963]). If $\overline{x}$ is a feasible solution to the primal problem (2.1) and $(\overline{y}, \overline{w})$ is a feasible solution to the dual problem (2.2) then $c\overline{x} \geq \overline{y}b + \overline{w}d$.*

Theorem 2.3.3 (Strong Duality Theorem) *(see [D1963]). If the primal problem (2.1) has an optimal solution x^* then the dual problem (2.2) has an optimal solution (y^*, w^*) and $cx^* = y^*b + w^*d$.*

Theorem 2.3.4 (Complementary Slackness Theorem) *(see [D1963]). A feasible solution $\overline{x}$ to the primal problem (2.1) and a feasible solution $(\overline{y}, \overline{w})$ to the dual problem (2.2) are optimal if and only if*

i) $\overline{x}_j > 0$ implies that $(\overline{y}, \overline{w})$ satisfies the corresponding dual constraint with equality, and

ii) $\overline{w}_i > 0$ implies that $\overline{x}$ satisfies the corresponding primal constraint with equality.

The conditions i) and ii) in Theorem 2.3.4 are called the *complementary slackness conditions.* This theorem shows that one easy way to prove some feasible primal solution is optimal for (2.1) is to provide a feasible dual solution for (2.2) which satisfies the complementary slackness conditions.

2.4 Basic Polyhedral Theory

In this section we define the basic terminology of polyhedral theory. Note that our discussion is brief and covers only what is essential for later sections. More detailed treatments of the subject can be found in [BG1982].

Given a set X of vectors, the *convex hull* of X, denoted by $Conv(X)$, is the set of all vectors w that can be written as a convex combination of vectors in S. That is, it is the set of all vectors w such that there exists non-negative scalars $\lambda_1, \lambda_2, \ldots, \lambda_k$ and vectors $v_1, v_2, \ldots, v_k$ in X such that $\sum_{i=1}^{k} \lambda_i = 1$ and $\sum_{i=1}^{k} \lambda_i v_i = w$.

A *polyhedron* $P \subseteq \mathbb{R}^E$ is the solution set of a finite system of linear equations and inequalities, and can be expressed in the form $P = \{x \in \mathbb{R}^E | Ax = b, Dx \leq d\}$. A *polytope* is a polyhedron which is bounded.

A subset F of a polyhedron P is a *face* of P if F is either the empty set or else the polyhedron obtained by taking the linear system which defines P and replacing some of the inequalities with the corresponding equations. A face F of P is called *proper* if $F \neq P$.

Any proper face of a non-empty polyhedron $P \subseteq \mathbb{R}^E$ is called a *vertex* if it consists of a single vector in $\mathbb{R}^E$. The set X of vertices of a polytope P has the property that $P = Conv(X)$. Thus optimizing some objective function cx over the polytope P is equivalent to optimizing cx over X.

Chapter 3

Finding a Convex Combination

Let z be any point in the fractional 2-factor polytope, Q_F^n, and let $\overline{Q}$ denote the complete set of vertices of Q_F^n. Since $z \in Q_F^n$, we can write z as a convex combination of a finite number of vectors in $\overline{Q}$. In this chapter, we present an algorithm for finding a finite subset Q of $\overline{Q}$ such that z is a convex combination of the vectors in Q, i.e., the algorithm provides a finite subset Q of $\overline{Q}$ such that we can write $z = \sum_{x \in Q} \lambda_x x$, where the λ_x's satisfy $\lambda_x > 0$ for all $x \in Q$ and $\sum_{x \in Q} \lambda_x = 1$.

Note that, in this chapter, we write A instead of A^t for convenience purposes whenever it is clear from the context that the transpose of A should be used.

Before we describe the algorithm, we give a constructive proof of Farka's Theorem [F1902]. This result is needed in the construction of the algorithm.

Theorem 3.0.1 (Farka's theorem) *Let A be an $n \times m$ real matrix and $b \in \mathbb{R}^n$. Then one or the other of the following statements is true.*

i. There is a solution $x \in \mathbb{R}^m$ such that $Ax = b$, $x \geq \mathbf{0}$.

ii. There is a $y \in \mathbb{R}^n$ such that $yA \geq \mathbf{0}$ and $yb < 0$.

Proof First, it is clear that both of these conditions cannot hold at the same time

since otherwise we obtain the contradiction $0 > yb = yAx \geq 0$. Now, suppose that condition i does not hold and consider the following LP:

$$\begin{aligned} &\text{maximize } \mathbf{0}x \\ &\text{subject to: } Ax = b \\ &\qquad x \geq \mathbf{0}. \end{aligned} \tag{3.1}$$

The dual of this LP is as follows:

$$\begin{aligned} &\text{minimize } yb \\ &\text{subject to: } yA \geq \mathbf{0}. \end{aligned} \tag{3.2}$$

According to duality theory, the infeasibility of (3.1) implies that the linear program given by (3.2) is either infeasible or unbounded. Since $\mathbf{0}$ is clearly a solution to $yA \geq 0$, we must have that LP (3.2) is unbounded.

Therefore, using the simplex method (see [D1948]) on the LP (3.2), we can obtain two vectors x^* and y^* such that both x^* and $x^* + ty^*$ are feasible solutions to $yA \geq 0$ for every $t \geq 0$, $t \in \mathbb{R}$ and such that $(x^* + ty^*)b$ can be made arbitrarily small by making t sufficiently large.

Consequently, we have that $(x^* + ty^*)A \geq 0$ for every $t \geq 0$, $t \in \mathbb{R}$. This implies that $y^*A \geq 0$. Moreover, since $(x^* + ty^*)b$ can be made arbitrarily small, we must have $y^*b < 0$.

The ray y^* therefore provides the vector y needed in condition ii of the theorem.□

3.1 Algorithm to find the representation

In this section, we present an algorithm which gives a representation of any point z in Q_F^n as a convex combination of a finite number of vertices of the fractional 2-factor polytope Q_F^n. This decomposition algorithm has previously been introduced in [RKPT2003] where it was applied to decompose any point in the Vehicle Routing Problem polytope as a convex combination of vertices of the TSP polytope. Here we

apply it to decompose any point $z \in Q_F^n$ as a convex combination of vertices of the fractional 2-factor polytope. The algorithm presented is subdivided into two phases.

3.1.1 Phase 1:

Start with any finite subset (perhaps empty) Q of $\overline{Q}$. Using linear programming, determine if z can be written as a convex combination of the vertices in Q, i.e., see if the following linear program has a feasible solution:

$$\begin{aligned} \text{maximize } & \mathbf{0}\lambda \\ \text{subject to: } & T\lambda = z \\ & \mathbf{1}\lambda = 1 \\ & \lambda \geq 0 \end{aligned} \tag{3.3}$$

where the columns of T represent the vertices in Q and $\lambda \in \mathbb{R}^Q$. If LP (3.3) has a feasible solution, then the vertices $x \in Q$ with $\lambda_x \neq 0$ give a representation of z as a convex combination of the points in Q and we are done. Otherwise, the constructive proof of Farka's Theorem provides a vector $y \in \mathbb{R}^{m+1}$, where m is the dimension of z, such that $y \begin{pmatrix} T \\ \mathbf{1} \end{pmatrix} \geq \mathbf{0}$ and $y \begin{pmatrix} z \\ 1 \end{pmatrix} < 0$. By letting $(w, p) \in (\mathbb{R}^m, \mathbb{R})$ be equal to y, we obtain $wz < p$ and $wx \geq p$ for every x in Q. We then go to Phase II.

3.1.2 Phase II:

Solve the following LP:

$$\begin{aligned} \text{minimize } & wy \\ \text{subject to: } & y \in Q_F^n \end{aligned} \tag{3.4}$$

where w is the vector in $\mathbb{R}^m$ provided in Phase I of the algorithm.

Since LP (3.4) will be optimized at some vertex of Q_F^n, the vector y^* which minimizes (3.4) will belong to $\overline{Q}$. Moreover, since $z \in Q_F^n$, we will have $wy^* \leq wz$. Combining this with the facts that $wz < p$ and $wx \geq p$ for all $x \in Q$ from above, we get that

$y^* \notin Q$. Thus, we can grow the set Q by adding y^* and return to Phase I.

When this algorithms terminates, we have a representation of z as a convex combination of vectors in Q, i.e., we have z written as a convex combination of a finite number of vertices of the fractional 2-factor polytope Q_F^n.

3.2 Speeding up the decomposition process

In this section, we present three enhancements which can be implemented to speed up the decomposition algorithm. The first one has previously been introduced in [RKPT2003] and the last two are new enhancements that we have contributed to the algorithm.

3.2.1 Enhancement 1: Fixing 0 and 1 edges

Lemma 3.2.1 *Let z be a point in Q_F^n. Any vertex x of Q_F^n present in the decomposition of z will satisfy:*

$$x_e = 1 \text{ for every edge } e \text{ such that } z_e = 1 \tag{3.5}$$

$$x_e = 0 \text{ for every edge } e \text{ such that } z_e = 0. \tag{3.6}$$

Proof For any edge e, we have $z_e = \sum_{x \in Q} \lambda_x x_e$ where $\sum_{x \in Q} \lambda_x = 1$ and $\lambda_x \geq 0$ for all x in Q. Moreover, since $0 \leq x_e \leq 1$ for every vector x in Q, if $1 = \sum_{x \in Q} \lambda_x x_e$ then we must have $x_e = 1$ for every vertex x with $\lambda_x \neq 0$. Similarly, if $0 = \sum_{x \in Q} \lambda_x x_e$, we must have $x_e = 0$ for every x with $\lambda_x \neq 0$. □

From the lemma above it follows that, given any point z in Q_F^n, we can facilitate the decomposition process by restricting the polytope Q_F^n in Phase II of the algorithm to the polytope defined by the set of points in Q_F^n which satisfy conditions (3.5) and (3.6) of Lemma 3.2.1.

By fixing all the zero and one edges of z, the only variables left in LP (3.4) of Phase II of the algorithm are the variables corresponding to the edges in the fractional support graph of z, i.e., the set of edges for which $0 < z_e < 1$. As will be shown in Chapter 6 when we discuss the computational results, these edges typically represent only a small fraction of the original set of edges. Hence, in practice, this restriction considerably reduces the search space of the LP solver.

For example, consider the point $z \in Q_F^{150}$ whose support graph is shown in Figure 3.1 (recall that the edges e not shown have $z_e = 0$). In Figure 3.2, we show the graph corresponding to the fractional support graph of z. It is easy to see that the edges in the fractional support graph of z represent only a small fraction of the original edge-set of z.

In practice, this enhancement also allows us to reduce the number of iterations needed for the decomposition process described earlier. If the zero and one edges of z are not fixed, then the vertex obtained in Phase II of the decomposition algorithm might not even satisfy the conditions of Lemma 3.2.1. If conditions (3.5) and (3.6) are not satisfied, then it is impossible to find a convex combination for z which contains this vertex. Hence, by fixing the zero and one edges, we increase the likelihood that the vertex obtained in Phase II of the algorithm does in fact belong in a convex combination for z, which should speed up the process.

3.2.2 Enhancement 2: Dividing z into smaller components

As shown in the section above, given a convex combination of the vector corresponding to the fractional support graph of z, which we denote by $\hat{z}$, we can find a convex combination for z. In particular, if $\hat{z}$ is given by $\sum \lambda_{\overline{x}} \overline{x}$ for some finite set of vectors then we have $z = \sum \lambda_{\overline{x}} x$ where, for a given vector $\overline{x}$, the corresponding vector x is given by

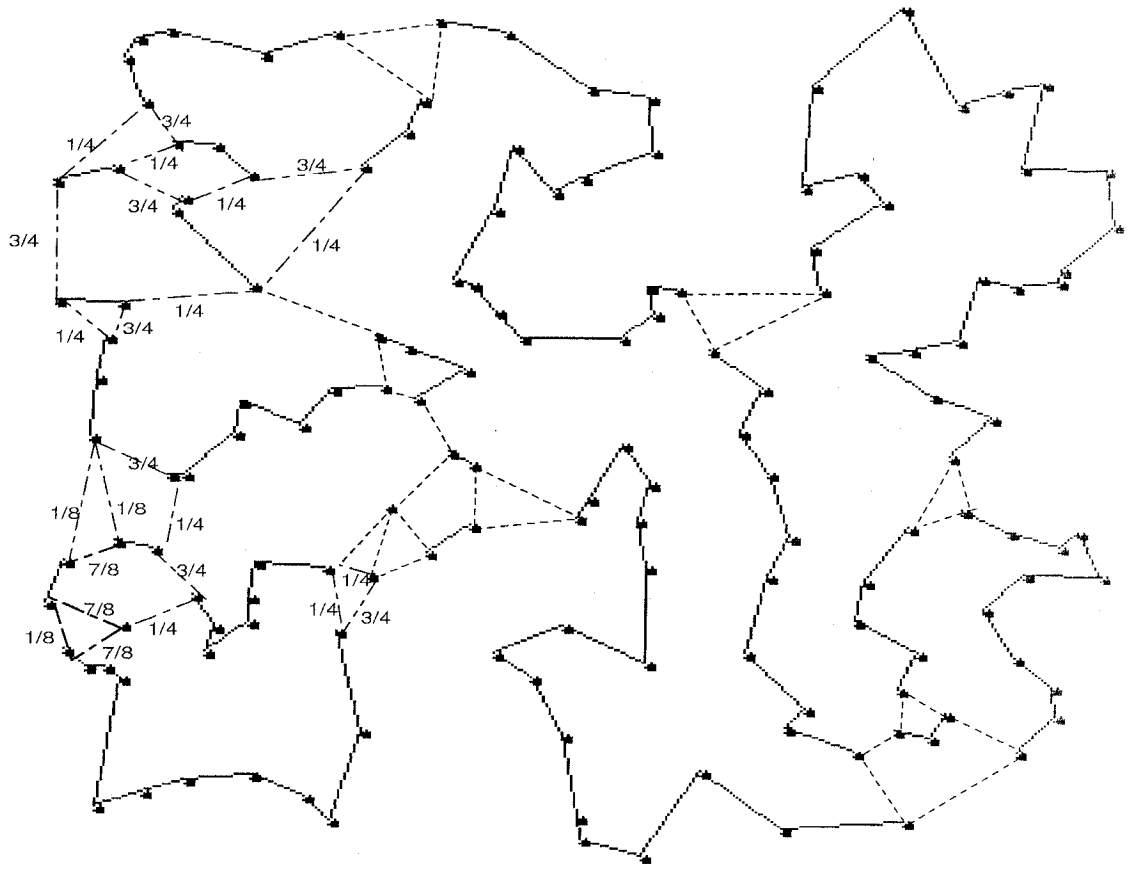

Figure 3.1: The support graph of a point in Q_F^{150}

$$x_e = \begin{cases} 0 & \text{if } z_e = 0, \\ 1 & \text{if } z_e = 1, \\ \overline{x}_e & \text{otherwise.} \end{cases}$$

Therefore, finding a convex combination for z amounts to finding a convex combination for $\hat{z}$.

The fractional support graph of z will be a graph with one or more connected components. Because of the structure of Constraints 1.11, finding the convex combination for the vector corresponding to any of these components can be done without consid-

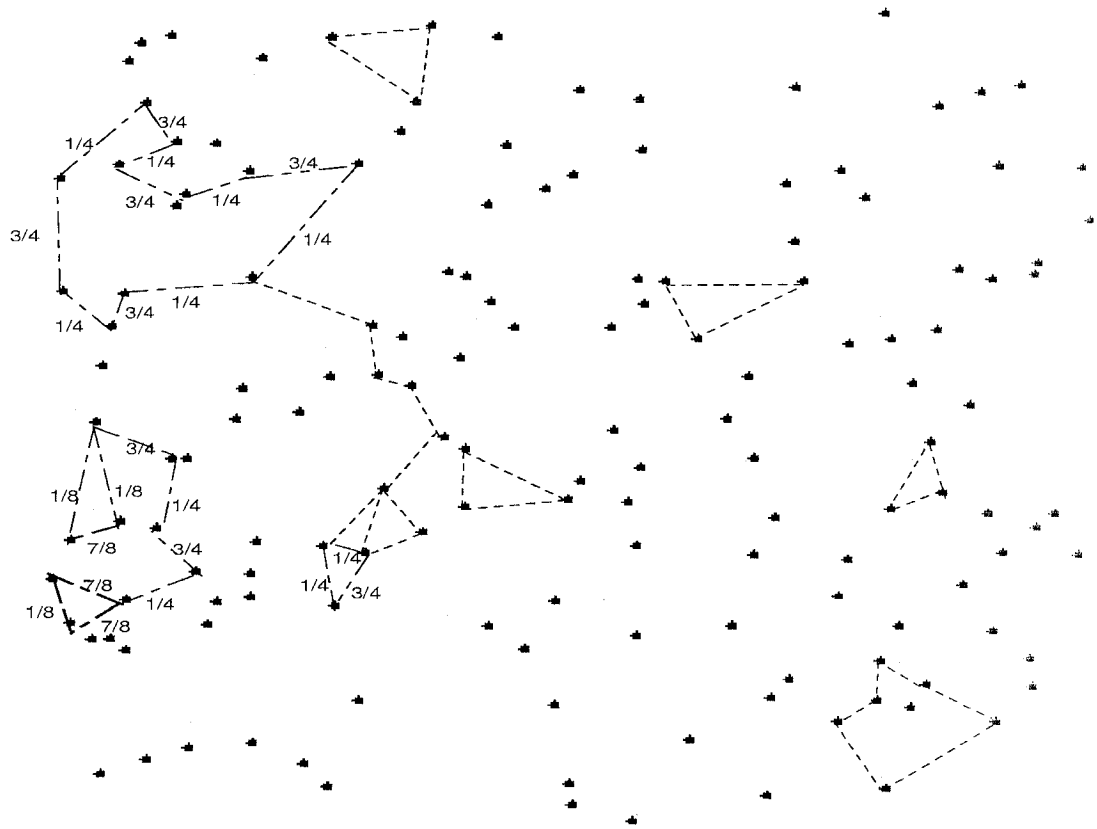

Figure 3.2: The fractional support graph of a point in Q_F^{150}

ering the value of the edges in any of the other components. Thus we can apply the decomposition algorithm presented earlier separately on each vector corresponding to the connected components of $\hat{z}$. This will reduce the size of the linear programs with which we work. Although we will now need to apply the decomposition process to each component, as we will demonstrate in Chapter 6, the amount of time saved by reducing the size of the LPs will be extremely significant in most examples.

In order to apply the decomposition algorithm to a vector corresponding to a component $C' = (V', E')$ of the fractional support graph of z, we will need to modify Phase I and Phase II of the algorithm in the following ways:

1. In Phase I, the set of vertices $\overline{Q}$ becomes the set of vertices of the polytope P described as follows:

$$P = \{x \in \mathbb{R}^{E'} : \sum_{uv \in E'} x_{uv} = 2 - \sum(1 : z_{uv} = 1, v \in V) \text{ for every node } u \in V' \text{ and } 0 \leq x_{uv} \leq 1 \text{ for every edge } uv \in E'\}. \quad (3.7)$$

2. In Phase II, LP (3.4) will have to be modified to

$$\begin{aligned} &\text{minimize } wy \\ &\text{subject to: } y \in P \end{aligned}$$

where P is the polytope described in (3.7).

Notice that we could even further reduce the time spent to find the convex combination for $\overline{z}$ by making use of several processors in parallel to compute a convex combination for each separate component.

Having found a set of vectors needed for the convex combination of each vector corresponding to the connected components of the fractional support graph of z, we can then use the following algorithm to construct a convex combination for our original vector z.

Algorithm to construct a convex combination for z

Let $z^{(1)}, z^{(2)}, \ldots, z^{(p)}$ represent vectors corresponding to the components of the fractional support graph of z. Suppose that each $z^{(i)}$ is written as

$$z^{(i)} = \sum_{j=1}^{n_i} \lambda_j^{(i)} w_j^{(i)}$$

where each $\lambda_j^{(i)} > 0$ for $1 \leq j \leq n_i$, $\sum_{j=1}^{n_i} \lambda_j^{(i)} = 1$ and we have $\lambda_j^{(i)} \leq \lambda_k^{(i)}$ whenever $j < k$.

The general idea behind the algorithm we have constructed is a greedy algorithm in which we repeatedly construct a vertex x of Q_F^n present in the convex combination

of z by considering, for each value of i between 1 and p, inclusively, the vector in the convex combination of $z^{(i)}$ which has the smallest non-zero coefficient. At each step of the process, the coefficient λ of the vertex x constructed will be the smallest possible coefficient among all non-zero $\lambda_j^{(i)}$, $1 \leq i \leq p$, $1 \leq j \leq n_i$. After subtracting λ from the coefficients of the vectors considered to construct x, this process will be repeated until z is written as a convex combination of vertices of the fractional 2-factor polytope Q_F^n. In essence we are repartitioning the values of the $\lambda_j^{(i)}$'s for each $z^{(i)}$, $1 \leq i \leq p$ (which sum to 1 for each $z^{(i)}$) so that we can match up the vectors w_j for the convex combination for z. This algorithm is detailed below.

Step 1 Initialize $y^{(k)} = 1$ for $k = 1, 2, \ldots, p$.

Let $\lambda_{\text{total}} = 0$.

Step 2 Let $\lambda = \min_{1 \leq i \leq p} \{\lambda_{y^{(i)}}^{(i)}\}$.

Step 3 Construct a vertex x of Q_F^n where

$$x_e = \begin{cases} 0 & \text{if } z_e = 0, \\ 1 & \text{if } z_e = 1, \\ w_{e\,y^{(k)}}^{(k)} & \text{otherwise} \end{cases}$$

where $1 \leq k \leq p$ represents the component to which edge e belongs.

Step 4 For each $1 \leq k \leq p$ substract λ from $\lambda_{y^{(k)}}^{(k)}$. If $\lambda_{y^{(k)}}^{(k)} = 0$, add one to $y^{(k)}$.

Step 5 Add x to the set of vertices in the convex combination of z and let its corresponding coefficient be $\lambda_x = \lambda$.

Step 6 Add λ to λ_{total}. If $\lambda_{\text{total}} = 1$, we are done. Otherwise, we go back to Step 2.

Note that in the above algorithm, each time we perform Step 4, at least one $\lambda_{y^{(k)}}^{(k)}$ becomes 0. Thus Steps 2 to 6 will be repeated at most $\sum_{i=1}^{p} n_i$ times.

As will be demonstrated in Chapter 6 when we discuss the computational results, our experiments have shown that by applying the decomposition process independently

to each vector corresponding to the components of the fractional support graph of z, not only do we reduce the amount of time needed to find a convex combination for z but we also considerably reduce the number of vertices present in the decomposition of z. As described in Chapter 1, the separation algorithm we are developing requires that we find the set of violated cut constraints in each one of the vertices present in the decomposition of z. Hence, by reducing the number of vertices in the convex combination, we also diminish the amount of time spent looking for violations.

However, as mentioned in Chapter 1, one of the main advantages of our algorithm for finding violations in z is that we can use several processors in parallel to find the violations in each one of the vertices present in the decomposition of z. Hence, if we work in a parallel environment, reducing the number of vertices in the convex combination of z will not have a direct impact on the amount of time needed. Nevertheless, it certainly affects the number of processors required. Consequently, reducing the number of vertices present in the decomposition of z is advantageous whether we work in a parallel or sequential environment.

3.2.3 Enhancement 3: Fixing odd disjoint half-cycles

Lemma 3.2.2 *Suppose that when we consider the fractional support graph of z, we find a component $C' = (V', E')$ which consists of an odd disjoint half-cycle, i.e., the edges in C' are all half-edges and they form a single odd length cycle. Then, for every edge e in E', we have $x_e = \frac{1}{2}$ for any vertex x present in the decomposition of z.*

Proof Using the decomposition algorithm described in this chapter to find a decomposition for C', the polytope P described in (3.7) will be the following:

$$P = \{x \in \mathbb{R}^{E'} : Ax = 1 \text{ and } 0 \leq x_{uw} \leq 1 \text{ for every edge } uv \in E'\} \qquad (3.8)$$

where A is the node-edge incidence matrix of C'. Hence the LP in Phase II of the algorithm will be of the form:

$$\begin{aligned} &\text{minimize } wy \\ &\text{subject to: } y \in P \end{aligned} \tag{3.9}$$

where P is the polytope given by (3.8). Since the length of C' is odd, the only vertex in P will be the vertex defined by $x_e = \frac{1}{2}$ for every edge e in C'. Since LP (3.9) will be optimized at a vertex of P, the result follows. □

Therefore, whenever we have a component C' which consists of an odd disjoint half-cycle, we know that the only vertex present in the decomposition of C' is the vector x defined by $x_e = \frac{1}{2}$ for every edge e in C'. Consequently, instead of using the decomposition algorithm to find a convex combination for C', we can obtain one directly using the previous result.

To illustrate the three enhancements described in this chapter, we will now construct a convex combination for the point z shown in Figure 3.1. As explained in Section 3.2.2, finding a convex combination for each vector corresponding to the connected components of the fractional support graph of z, shown in Figure 3.2, will allow us to construct a convex combination for z. Moreover, as mentioned in Section 3.2.3, the edges which form odd disjoint half-cycles in Figure 3.2 have value $\frac{1}{2}$ in every vertex present in the decomposition of z. Hence, we only need to find a convex combination for three of the components of the fractional support graph of z. In Figure 3.3, we show a convex combination for the vectors corresponding to these three components. Following the algorithm described in Section 3.2.2, we can construct a convex combination for the point z. In Figure 3.4, we have drawn the six vertices present in the decomposition of z obtained using this algorithm. The coefficient λ of the first five vertices is $\frac{1}{8}$ and the coefficient of the last vertex drawn is $\frac{3}{8}$.

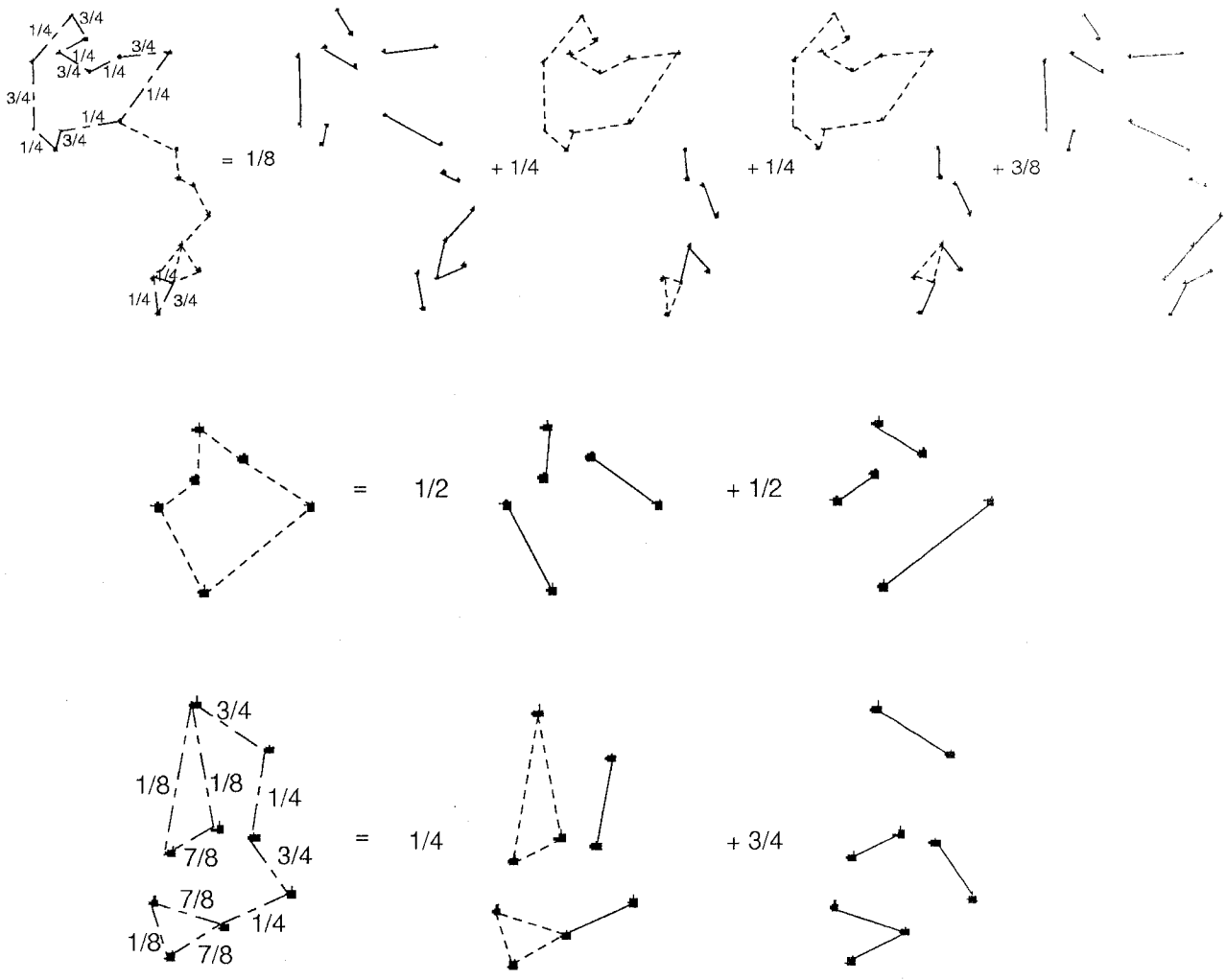

Figure 3.3: A convex combination for three of the components of the graph drawn in Figure 3.2

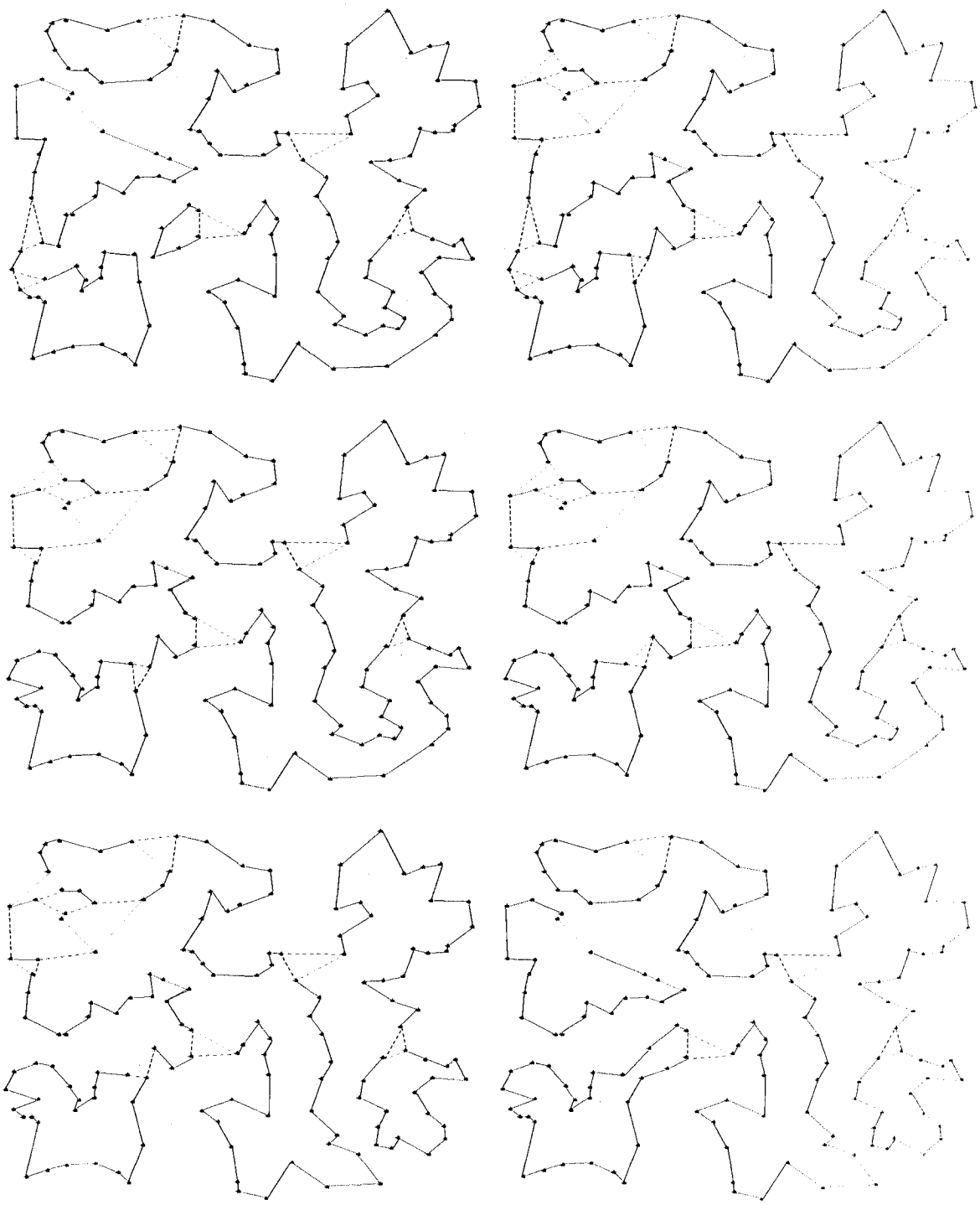

Figure 3.4: Vertices in the convex combination of the point drawn in Figure 3.1

3.3 Extending the decomposition algorithm to other combinatorial optimization problems

The decomposition algorithm described in Section 3.1 can be applied in a straightforward manner to other combinatorial optimization problems. In particular, whenever we have a polytope P included in another polytope P' and we have an efficient algorithm for finding the violations in the vertices of the polytope P' then we can use the algorithm described in this chapter to express any point in P as a convex combination of vertices of the larger polytope P'. Using this decomposition algorithm, we can then use the results of Lemma 1.0.1 and Corollary 1.0.2 of Chapter 1 to find violations in any point belonging to P (see Chapter 1).

Moreover, whenever the polytope P' is of the form $\{x \in \mathbb{R}^E; Ax = b, l \leq x \leq u\}$ where A is the node-edge incidence matrix of K_n, then the three enhancements described in this chapter can easily be transferred to help speed up the decomposition process for the points in P.

Chapter 4

Finding Violated Subtour Elimination Constraints in any Vertex of the Fractional 2-Factor Polytope

Given $K_n = (V, E)$ and a vector $c \in \mathbb{R}^E$ of edge costs, the *Fractional 2-Factor Problem* consists of finding $x \in \mathbb{R}^E$ which minimizes cx subject to x satisfying the degree constraints (1.5) and the lower-bound and upper-bound constraints (1.7) defined in Chapter 1 for Q_T^n. Therefore the Fractional 2-Factor Problem amounts to solving:

$$\begin{aligned} &\text{minimize } cx && \\ &\text{subject to: } x(\delta(v)) = 2 && \text{for all } v \in V \\ &0 \leq x_e \leq 1 && \text{for all } e \in E. \end{aligned} \tag{4.1}$$

The corresponding polytope, which we denote by Q_F^n, is defined by $Q_F^n = \{x \in \mathbb{R}^E | Ax = 2, 0 \leq x \leq 1\}$, where $A \in \mathbb{R}^{V \times E}$ is the node-edge incidence matrix of K_n. As described in the next lemma, the vertices of this polytope have a very special structure (see [B1970]).

Lemma 4.0.1 *Any vertex $\hat{x}$ of Q_F^n is half-integer, and the set of edges $J \subseteq E$ such*

that $\hat{x}_j = \frac{1}{2}$ *for all* $j \in J$ *partitions into the edge-sets of an even number of odd disjoint cycles.*

Let z be any point in Q_F^n. Since z belongs to the fractional 2-factor polytope, we can write z as a convex combination of vertices of this polytope. As explained in Chapter 1, in order to find the complete list of cut constraints violated by z, it suffices to find the complete set of violations in each of the vertices present in the decomposition of z and then check each one to see if it is violated by z. However, as will be discussed in this chapter, a vertex of Q_F^n can have an exponential number of violated cut constraints and verifying each one of those constraints in z can be impractical or even impossible with today's technology. Hence, given a vertex x of Q_F^n, instead of constructing an algorithm which finds the complete list of violations in x, we will instead restrict ourselves to a reasonable number of violations. By doing so, we should find a good portion of the violations in z. Moreover, as will be shown in this chapter, the algorithm we develop is guaranteed to find at least one violation in z if one exists. Therefore, we have an exact separation algorithm.

Separation algorithm

Let x be a vertex of the fractional 2-factor polytope, Q_F^n, and let G_x denote the support graph of x. Since the half-edges of G_x form disjoint cycles (see Lemma 4.0.1), the number of half-edges contained in $\delta(S)$ must be even for any subset $S \subset V$. Moreover, since every non-zero edge in G_x has value $\frac{1}{2}$ or 1, the value of any cut violated by x must either be 0 or 1.

Before we describe the separation algorithm, we first look at a very efficient heuristic to help reduce the size of the vertices with which we work.

Contracting out paths consisting of edges of weight one

In this section we define a 1-path in the support graph of x to be any path P in G_x which satisfies $x_e = 1$ for every edge $e \in P$.

Given any vertex x of the fractional 2-factor polytope, before we start looking for violations in x, we first contract the 1-paths in G_x to single 1-edges. As illustrated in Figures 5.1 and 5.2, this process might create multiple edges. Note that the 1-paths which consist of a cycle are contracted to a single node.

By contracting the 1-paths to 1-edges, we reduce the dimension of the vertices with which we work and, in turn, this will help speed up the separation algorithm. Note that shrinking the 1-path does not create addition violations and that every violation in the original graph corresponds to a violation in the contracted graph. Now consider any violation in the contracted graph. If this violation does not cross any 1-edges, then it corresponds to a unique violation in the original graph. On the other hand, if this violation does cross a 1-edge e then for every 1-edge in the 1-path represented by e, there exists a corresponding violation in the original vertex which crosses this 1-edge.

Thus, finding the complete set of violations in any vertex x of the fractional 2-factor polytope amounts to finding the complete set of violations in the vertex obtained by contracting out the 1-paths in G_x to single 1-edges. Since the dimension of the contracted vertex corresponding to x is, in practice, smaller than the dimension of x, we have decided to include this heuristic in our separation algorithm. In the remainder of this chapter, whenever we refer to a vertex x of the fractional 2-factor polytope, we assume that the 1-paths in G_x have already been contracted to 1-edges. However, for every violation we find which crosses a 1-edge in the contracted vertex, we must remember to consider the complete set of corresponding violations in the original vertex.

4.1 Finding all cuts of value 0

If x has a cut of value 0, i.e., if $x(\delta(S)) = 0$ for some subset $S \subset V$, then G_x is disconnected and the subgraph of G_x induced by the nodes in S corresponds to a set of connected components of G_x. Conversely, if G_x is disconnected, then each connected component of G_x defines a violated subtour constraint in x. In fact, any set of connected components defines a violated constraint, giving a number of violated constraints exponential in the number of components. Hence, if the number of connected components in G_x is relatively big then we cannot possibly hope to report every violation. To attempt to overcome this problem, we will only consider the constraints defined by one component. Note that by doing so, if G_x contains more than three components, we will be overlooking some violations in x and hence we will not be guaranteed to find every violation in z.

For example, consider the point $z \in Q_F^{12}$ whose support graph is shown in Figure 4.1. One possible decomposition of z would be $\frac{1}{3}x_1 + \frac{2}{3}x_2$, where the support graphs corresponding to x_1 and x_2 are shown in Figure 4.2 a) and b), respectively. We can easily verify that z satisfies the cut constraints corresponding to each connected component of G_{x_1} and G_{x_2}. However, if we let S_1 be equal to the node-set $\{1, 2, 3\}$ and S_2 be equal to the node-set $\{4, 5, 6\}$ then the constraint corresponding to the cut $\delta(S_1 \cup S_2)$ is violated by z.

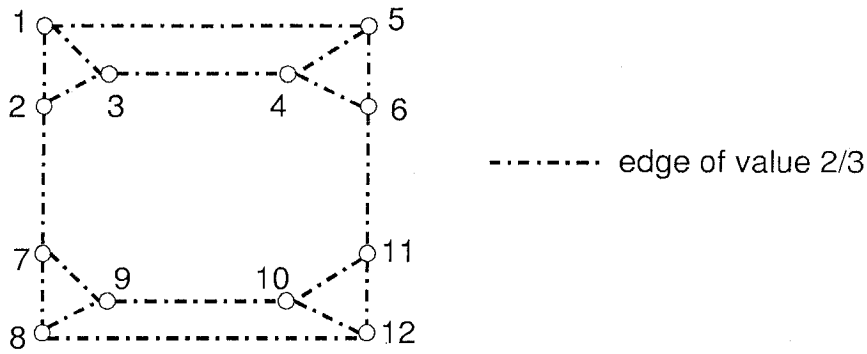

Figure 4.1: G_z for a point $z \in Q_F^{12}$

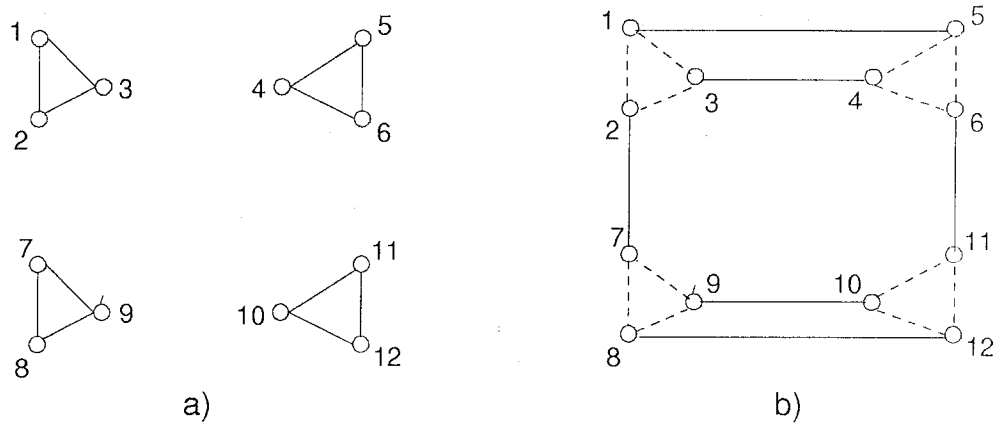

Figure 4.2: Support graphs for two vertices x_1 and x_2 of Q_F^{12}

Recall that for every violated cut $\delta(S)$ found in x we have to check the associated constraint in z, i.e., we have to compute $z(\delta(S))$. Hence it does not make sense to generate an exponential number of violated constraints in x and our restriction to the set of constraints defined by one component of G_x is justified. However, if the only cut constraint violated in z is a constraint corresponding to a set of connected components in G_x then this cut might remain hidden even after we consider every vertex in the decomposition of z. To eliminate this problem, we will use any fast minimum cut algorithm to find the minimum cut in the complete weighted graph on p nodes, K_p, defined as follows: p is the number of connected components in G_x and

$K_p = G_z \circ S_1 \circ S_2 \circ \ldots \circ S_p$ where G_z denotes the support graph of the point z and $S_1, S_2, \ldots, S_p$ are the node-sets of the connected components of G_x. Note that if the process of shrinking the graph G_z creates multiple edges, then we simply merge then and add their weight values. If the minimum cut $\delta(S)$ in K_p has value less than 2, then the nodes in S represent the connected components of G_x whose union form a violation in z, i.e., the cut $\delta(S')$ where $S' = \bigcup_{U \in S}\{v \in V | v$ belongs to component U in $G_x\}$ corresponds to a violated cut in z. On the other hand, if the minimum cut in K_p has value greater than or equal to 2, then no set of connected components of G_x corresponds to a violated cut in z. Thus, for every vertex x present in the decomposition of z, if G_x has more than three components, we will construct the corresponding complete graph K_p and use any fast minimum cut algorithm to find the minimum cut in K_p. Consequently, if it is possible to find at least one violation in z when we consider every set of connected components in G_x, then finding the minimum cut in K_p will allow us to find one of these violations.

To illustrate this fact, consider the point z and the vertex x_1 of Q_F^{12} whose support graphs are shown in Figure 4.1 and Figure 4.2 a), respectively. To verify if any set of connected components of G_{x_1} corresponds to a violation in z, we will construct the corresponding graph K_p described beforehand. Since G_{x_1} has four connected components, we have $p = 4$. Moreover, if we let S_1 be equal to the node-set $\{1, 2, 3\}$, S_2 be equal to the node-set $\{4, 5, 6\}$, S_3 be equal to the node-set $\{7, 8, 9\}$ and S_4 be equal to the node-set $\{10, 11, 12\}$, then we have $K_4 = G_z \circ S_1 \circ S_2 \circ S_3 \circ S_4$. This graph is illustrated in Figure 4.3. By computing the minimum cut in this graph, we find a cut of value $\frac{4}{3}$ given by the two nodes corresponding to the node-sets S_1 and S_2. Therefore, the cut $\delta(S_1 \cup S_2)$ is a violated cut in z.

Many fast minimum cut algorithms exist, ranging from algorithms that follow from the early work on maximum flows in the 1950s [GH1961] to a more recent Monte Carlo randomized algorithm that runs in $O(m \log^3 n)$ time on a graph with m edges and n

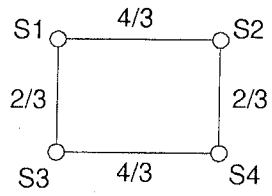

Figure 4.3: The graph K_4 correponding to a vertex of Q_F^{12}

nodes [K1996]. Since every component in a vertex of Q_F^n can have as few as three nodes, the number of components in a vertex of Q_F^n can be as large as $\frac{n}{3}$. Thus, in theory, we could be solving a minimum cut problem on $\frac{n}{3}$ nodes. However, as will be discussed in Chapter 6, in practice, the number of connected components in a vertex of the fractional 2-factor polytope is relatively small compared to the number of nodes in the vertex and thus the time needed to construct the graph K_p and compute the minimum cut in this graph is, in most cases, negligible.

4.2 Finding all cuts of value 1

If x has a cut $\delta(S)$ of value 1, then the edges in $\delta(S)$ either consist of a single edge of value 1 or two half-edges belonging to the same half-cycle in x. This observation follows directly from the properties of the vertices of the fractional 2-factor polytope (see Lemma 4.0.1). In this section, we say that a cut is of *type A* if it consists of a single edge of value 1 and that a cut is of *type B* if it consists of two half-edges of the same half-cycle in x.

4.2.1 Finding the set of cuts of type A

If a cut in x consists of a single edge of value one then this edge must correspond to a cut edge in G_x, i.e., an edge whose removal disconnects G_x. Such edges can easily be detected by following the well-known algorithm described below.

Algorithm 4.2.1 (Algorithm to find cut edges in a graph G)

Step 1. Perform a depth first search visit of G numbering the nodes in G in ascending order as they are visited. Let *Num(v)* be the number assigned to a node v of $V(G)$.

Step 2. For each edge $uv \in E(G)$ not appearing in the depth first search (DFS) tree obtained, add a "back edge" uv to the tree.

Step 3. Considering the nodes in $V(G)$ in the reverse order they were visited in Step 1, for every node v in $V(G)$, define

$$\text{low}(v) = \min \begin{cases} \text{num}(v) \\ \min\{\text{num}(w) | vw \text{ is a back edge of the DFS tree } \} \\ \min\{\text{low}(w) | w \text{ is a child of } v \text{ in the DFS tree}\} \end{cases}$$

Step 4. If $\text{low}(v) = \text{num}(v)$ for any node $v \in V(G)$ then the edge vw where w is the parent node of v in the DFS tree is a cut edge in G.

This algorithm allows us to find every cut of type A in x.

4.2.2 Finding the set of cuts of type B

The cuts which consist of two half-edges are more difficult to identify. In this section, we give an efficient algorithm for finding all such cuts.

Since every cut of type B consists of two half-edges belonging to the same half-cycle in x, we would like to find a way to work with each half-cycle independently. Consequently, the main idea behind the algorithm we are constructing is to modify the graph G_x in such a way as to be able to work with each half-cycle independently. To do so, we first need the following result.

Lemma 4.2.1 *Let C be any half-cycle in G_x and let u and v be two nodes in C. If there exists a uv-path p in G_x which does not use any edges in C, then any uv-cut in x will have a value greater than or equal to 2 and hence, will not be of any interest to us.*

Proof The uv-path p will consist of edges of value 1 and $\frac{1}{2}$. If a cut crosses a half-edge, it must also cross another half-edge of the same half-cycle (see Lemma 4.0.1). Hence, any cut which crosses the path p will have a value greater or equal to 1. Since any uv-cut will have to at least cross p and two half-edges in C, the value of any uv-cut will be at least 2. □

From this result, we conclude that if, for a given pair of nodes u and v in C, there exists a uv-path in G_x not using any edges in C, then the two nodes u and v must belong to the same side of any violated cut in x. In the next section, we describe an algorithm for finding the pairs of nodes u, v in C for which there exist a uv-path in G_x not using any edges in C.

Algorithm for constucting a graph H' which will allow us to work with each half-cycle independently

In this section, we describe an algorithm which constructs a weighted multigraph H' allowing us to work with each half-cycle independently. The main idea behind the algorithm is as follows. Let C be any half-cycle in G_x and let $G_x \setminus E(C)$ denote the graph obtained by removing from the support graph of x the edges in C. Consider two nodes u and v in C. If u and v belong to the same component of $G_x \setminus E(C)$, then there exists a uv-path in G_x which does not use any edges in C. Hence, by the previous result, we know that these two nodes must belong to the same side of any violated cut in x. As we will demonstrate in Lemma 4.2.2, joining the nodes in C which belong to the same component of $G_x \setminus E(C)$ by a 1-path in H' will allow us to work with each half-cycle independently. Moreover, we also know that every node in

V which belongs to the same component of $G_x \setminus E(C)$ will have to belong to the same side of any violated cut of type B in x which crosses the half-cycle C. Therefore, since we want to work with each half-cycle independently, for every connected component of $G_x \setminus E(C)$, we are going to choose a node j in C from this component and insert in attached_node[j] a list of every node in V which belongs to this component. Hence, later, when we work with each half-cycle independently, for every cut of type B found which crosses C, we will use the data-structure 'Attached_node' to quickly obtain the corresponding cut of type B in x, i.e., if the cut $\delta(S)$ is a cut of type B in C, then the cut $\delta(S^*)$ where $S^* = \bigcup_{u \in S}\{\text{Attached\_node}[u]\}$ is the corresponding cut of type B in x.

The algorithm is detailed below.

Algorithm 4.2.2

Step 1. (Initialization) For every node j in V, let Attached_node[j] be equal to the empty list. Define a new weighted multigraph $H' = (V^*, E^*)$ where $V^* = V$ and $E^* = \{e \in E | x_e = \frac{1}{2}\}$ with associated weight function h and define $h_e = \frac{1}{2}$ if edge e belongs to E^* and 0, otherwise.

Step 2. For every half-cycle, C, in G_x:

i. Consider the graph $G_x \setminus E(C)$ obtained by removing from the graph G_x the edges in C.

ii. Using depth first search, find the connected components of $G_x \setminus E(C)$.

iii. For every set of nodes $\{u_1, u_2, \ldots, u_p\}$ in C which belong to the same component of $G_x \setminus E(C)$ and for every $i = 1, 2, \ldots, p-1$, add an edge of value 1 in H' joining node u_i and u_{i+1} (note that the order in which the nodes $u_1, u_2, \ldots, u_p$ are considered is not important). Insert in Attached_node[u_1] the complete list of nodes in V which belong in this connected component.

Since the half-cycles in G_x are disjoint and since every half-cycle has at least three nodes, there will be at most $\frac{n}{3}$ half-cycles in G_x. Moreover, since there are $O(|V|)$ edges in G_x, finding the connected components of $G_x \setminus E(C)$ can be done in $O(|V|)$ time using depth first search. It is thus clear that the above algorithm can be implemented in $O(|V|)$ time when working in a parallel environment with $O(|V|)$ processors.

In the figure below we show the support graph of a vertex x of the fractional 2-factor polytope. Figure 4.5 illustrates the multigraph H' corresponding to the vertex x obtained using Algorithm 4.2.2.

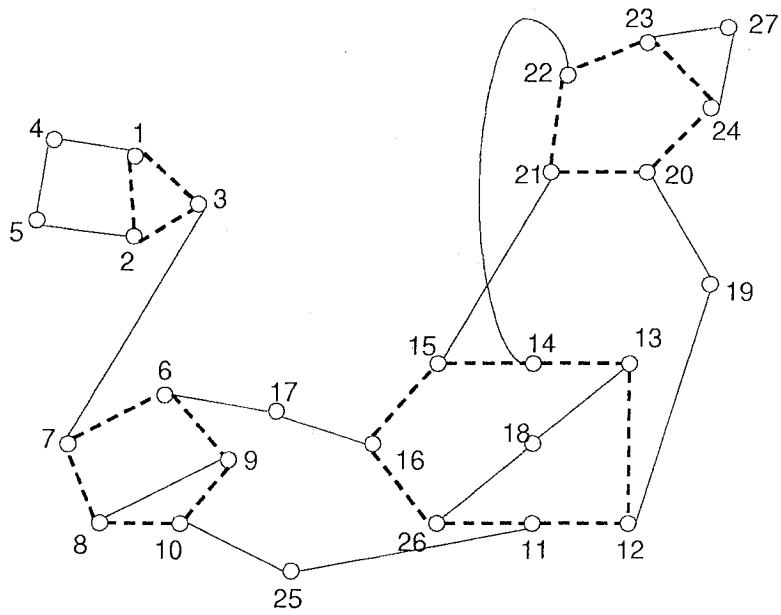

Figure 4.4: The support graph of a vertex of Q_F^{27}

We now show that finding the violations of type B in x is equivalent to finding the violations of type B in each component of H'.

Lemma 4.2.2 *Let H' be the multigraph on n nodes obtained using Algorithm 4.2.2 and consider the weight vector h' corresponding to this graph. Let e_1 and e_2 be two half-edges belonging to the same half-cycle C in G_x. Let $\delta(S_1)$ be the minimum cut in*

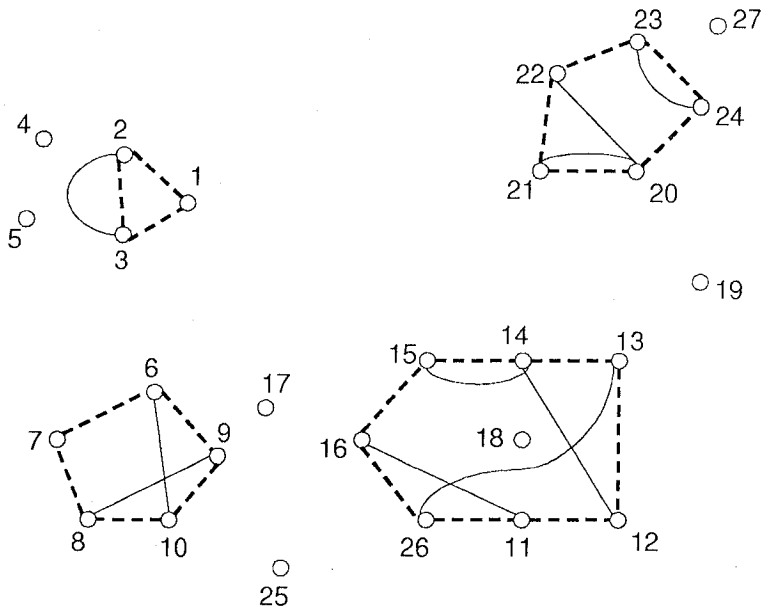

Figure 4.5: The graph H' corresponding to the graph shown in Figure 4.4

H' which crosses these two edges and let $\delta(S_2)$ be the minimum cut in G_x which crosses these two edges. We have $h'(\delta(S_1)) < 2$ if and only if $x(\delta(S_2)) < 2$.

Proof First suppose that $h'(\delta(S_1)) \geq 2$. Let $\delta(S_3)$ be any cut in x which crosses e_1 and e_2. Since $h'(\delta(S_1)) \geq 2$, we must have $h'(\delta(S_3)) \geq 2$. If $\delta(S_3)$ crosses four or more half-edges in G_x, then it is clear that $x(\delta(S_3)) \geq 2$. If $\delta(S_3)$ only crosses the two half-edges e_1 and e_2, then since $h'(\delta(S_3)) \geq 2$, the cut $\delta(S_3)$ must also cross a 1-edge xy in H' joining two nodes x and y in C. By construction of the graph H', these two nodes belong to the same component of $G_x \setminus E(C)$ and hence, there exists a xy-path in G_x not using any edges in C. Using the result of Lemma 4.2.1, we conclude that $x(\delta(S_3)) \geq 2$. Since $\delta(S_3)$ was arbitrary, we get that $x(\delta(S_2)) \geq 2$.

Conversely, suppose that $x(\delta(S_2)) \geq 2$. By setting the value of x_{e_1} and x_{e_2} to 0 in x, the value of the minimum cut which crosses the edge e_1 and e_2 in the resulting graph will be greater than or equal to 1. Thus, if we let u and v be the two adjacent nodes of edge e_1, then there exists a uv-path p^* in G_x which does not use the edges

e_1 and e_2. By removing the two edges e_1 and e_2 from C, we partition the nodes in C into two sets C_1 and C_2. Since, the path p^* cannot consist entirely of edges in C, there must exist a subpath p_1 of p^*, with end nodes p and q, such that $p_1 \cap C = \emptyset$ and such that p and q belong to C_1 and C_2, respectively. Therefore, nodes p and q belong to the same component of $G_x \setminus E(C)$. From the construction of the graph H', there exists a 1-path p^* joining nodes p and q in H'. Now consider any cut $\delta(S_3)$ in H' which crosses e_1 and e_2. If this cut does not cross any other edge in C then it must cross at least one edges from the 1-path p^*. Hence, it will have a value greater than or equal to 2. Moreover, any cut in H' which crosses e_1, e_2 and some other edge in C will also have a value greater than or equal to 2. Hence, the result follows. □

Consequently, if we are only interested in finding the violations of type B in x, it suffices to construct the corresponding graph H' using Algorithm 4.2.2 and find the violations of type B in H'. The graph H' will consist of several disconnected half-cycles with possibly some other 1-edges joining nodes belonging to the same half-cycle. Hence, finding the complete list of violations of type B in H' is equivalent to finding the complete list of violations of type B in each component of H'. We can thus work independently with each half-cycle of x using several processors in parallel. Therefore, in order to find the complete list of violations of type B in H', we require an algorithm which is able to find all the violations of type B in any graph consisting of a half-cycle and possibly some other edges of value 1 joining the nodes of the half-cycle.

Algorithm to find violations of type B in any connected component of the graph H'

Any connected component $G = (V', E')$ with associated weight function g of the graph H' will consist of a half-cycle C and possibly some other 1-edges joining the nodes in C. Moreover, recall from the construction of the graph H', that every node in G will have at most two adjacent 1-edges.

In this section we develop an $O(|V'|)$ algorithm for finding the complete set of cut violations of type B in any connected component $G = (V', E')$ of H'.

Let $G = (V', E')$ be any connected component of H' with associated weighted function g and let C represent the half-cycle in G. Let C be written as $(u_1, u_2, u_3, \ldots, u_{|V'|})$. Before we describe the algorithm, we will label the nodes $u_1, u_2, \ldots, u_{|V'|}$ in C as $1, 2, \ldots |V'|$, respectively. Moreover, in the remainder of this section, whenever we refer to node i in G, we refer to the node in G which has label i.

Definition Let uv and xy be two non-adjacent 1-edges in G with $u < v$ and $x < y$. We say that these two 1-edges *cross* if we either have $u < x < v < y$ or $x < u < y < v$.

Using the above definition, it is easy to see that the two 1-edges in the half-cycles shown in Figure 4.6 cross.

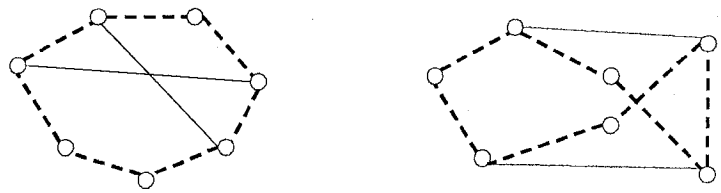

Figure 4.6: Two examples of half-cycles with crossing 1-edges

Lemma 4.2.3 *If two 1-edges uv and xy in G are crossing or adjacent to one other then the nodes u, v, x and y must belong to the same side of any violated cut in G.*

Proof It is clear that nodes u and v must belong to the same side of any violated cut in G. Moreover, the same is true for nodes x and y. If these two edges are adjacent, then the result follows directly. Otherwise, suppose that $x \neq u$ and consider any ux-cut in G. Either this cut crosses four or more half-edges or it crosses two half-edges and one or both of the 1-edges uv and xy. Therefore, any such cut will have value greater or equal to 2. Hence, node u and x must belong to the same side of any violated cut in G and the result follows. □

Corollary 4.2.4 *Let S_1 and S_2 be two subsets of nodes in G. Suppose that every node in S_1 belongs to the same side of any violated cut in G and assume the same assumption is also true for the set S_2. If a 1-edge joining a pair of nodes in S_1 crosses or is adjacent to a 1-edge joining a pair of nodes in S_2, then the nodes in $S_1 \cup S_2$ must all belong to the same side of any violated cut in G.*

Proof This result follows directly from the above lemma. □

In the next section, we describe an algorithm which allows us to find the pairs of crossing 1-edges in any connected component G of the graph H' constructed earlier. This algorithm can thus be used to find which set of nodes in G belong to the same side of any violated cut. This linear time algorithm is based on an algorithm developed by Hopcroft and Tarjan which finds separating pairs of nodes in graphs (see [HT1973]).

Algorithm to find pairs of crossing 1-edges in any connected component G of the graph H'

In this algorithm, we define a new graph H which is used to keep track of adjacent 1-edges and pairs of crossing 1-edges. The nodes in H are the unordered pairs (u, v) where uv is a 1-edge in G. At any point during the algorithm, if we find two 1-edges ab and cd in G which cross or are adjacent to one other, then we will add an edge in H from node (a, b) to node (c, d) to indicate that this pair either cross or are adjacent. In order to identify such crossing pairs, we will keep a stack (called TSTACK) of triples (h, a, b) where the pair $\{a, b\}$ is an 1-edge in G and the variable h is an extra variable needed to keep track of previous crossings found. The pairs $\{a, b\}$ are in nested order on the stack, i.e., at any point during the algorithm, if $(h_1, a_1, b_1), (h_2, a_2, b_2), \ldots, (h_k, a_k, b_k)$ is the current stack content, then $a_k \leq a_{k-1} \leq \ldots \leq a_2 \leq a_1 \leq b_1 \leq b_2 \leq \ldots \leq b_k$. Moreover, we also have $h_1 \leq h_2 \leq \ldots \leq h_k$. These properties of TSTACK will allow us to quickly identify crossings and adjacent 1-edges. The algorithm is detailed below.

Algorithm 4.2.3

Step 1. (Initialization) Construct a graph H where the nodes of H are the unordered pairs (u, v) where uv is a 1-edge in G. For every node j in G, define

$$\text{highpt}(j) = \max\{i | ij \text{ is a 1-edge in } G\}.$$

If node j has no adjacent 1-edges then we define highpt(j) to be equal to 0.

Step 2. For $j = |V'|, \ldots, 1$,

A. For every 1-edge ij in G with $i < j$ (if two such edges exist, choose the one which has the largest value of i first) :

a) Let $h^* = j$.

b) While the triple (h, a, b) on top of TSTACK satisfies $a > i$ (Note: This means that edge ij crosses or is adjacent to edge ab, see Lemma 4.2.6).

i. Add an edge joining node (a, b) and node (i, j) in H.

ii. Let $h^* = h$.

iii. Pop the triple (h, a, b) from the TSTACK.

c) Push the triple (h^*, i, j) on TSTACK.

B. While the triple (h, a, b) on top of TSTACK satisfies highpt$(j) > h$ (Note: This means that edge $(j, \text{highpt}(j))$ crosses or is adjacent to edge ab, see Lemma 4.2.7).

a) Add an edge joining node (a, b) and node $(j, \text{highpt}(j))$ in H.

b) Pop the triple (h, a, b) from the TSTACK.

C. While the triple (h, a, b) on top of TSTACK satisfies $a = j$.

a) Pop the triple (h, a, b) from TSTACK.

Step 3. For all adjacent pairs of 1-edges ab and cd in G, add an edge joining nodes (a, b) and (c, d) in H if these two nodes do not already belong in the same

component of H. (Note that this step is not necessary and we only include it to simplify the proofs concerning this algorithm.)

Consider the graph drawn in Figure 4.7. If we apply Algorithm 4.2.3 to this graph, then the graph H obtained is the graph shown in Figure 4.8.

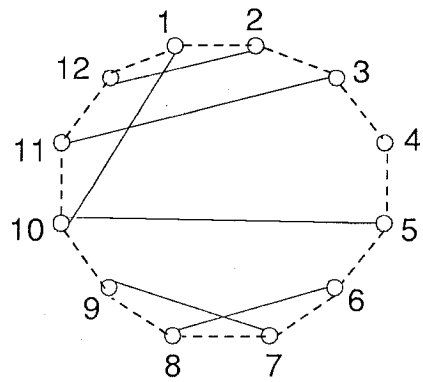

Figure 4.7: An example of a possible component of H'

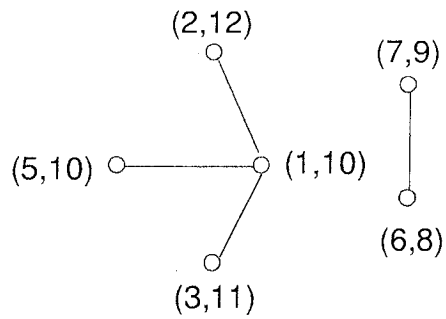

Figure 4.8: A graph H obtained using Algorithm 4.2.3

Before we prove the correctness of this algorithm, we first need to introduce some notation. Define the *h-value* of a node (x, y) in H as the first entry of the triple inserted on TSTACK when edge xy is considered in Step 2A of the algorithm. Since every 1-edge in G is considered once and only once, the h-value of a node (x, y) in H is well-defined. Moreover, if C represents a component of H at the end of the

algorithm, then we define $h(C) = \max\{h\text{-value}(x,y)|(x,y) \in C\}$. We also define $X(C) = \min\{x|(x,y) \in C$ and $h\text{-value}(x,y) = h(C)\}$ and $Y(C) = min\{x|(x,y) \in C\}$.

Before, we present the main results proving the correctness of this algorithm, we first look at some preliminary lemmas.

Lemma 4.2.5 *At any point during the algorithm, if* $(h_1,a_1,b_1),(h_2,a_2,b_2),\ldots,(h_k,a_k,b_k)$ *are the contents of TSTACK from top to bottom, then* $a_k \leq a_{k-1} \leq \ldots \leq a_2 \leq a_1 \leq b_1 \leq b_2 \leq \ldots \leq b_k$. *Moreover, we have* $h_1 \leq h_2 \leq \ldots \leq h_k$.

Proof From the algorithm, triples are only added to TSTACK in Step 2A c). By the condition of Step 2A, the triple (h^*,i,j) is only added to TSTACK if $i < j$. Moreover, this triple will not be added until the current triple (h,a,b) on top of the stack has $a \leq i$. Therefore, we must have $a_k \leq a_{k-1} \leq \ldots \leq a_1 < b_1$. Also, since the b value of any triple (h,a,b) corresponds to the j value when that triple was added to TSTACK, it follows from the fact that we look at the j values in decreasing order that $b_1 \leq b_2 \leq \ldots \leq b_k$.

It therefore remains to be shown that $h_1 \leq h_2 \leq \ldots \leq h_k$. For each new value of j in Step 2 of the algorithm, we initially have $h^* = j$, and since the j values are considered in decreasing order, we must have that h^* is less than or equal to all h_i values for triples (h_i,a_i,b_i) below it in the stack. Since the value of h^* can only decrease to h_i as the triple (h_i,a_i,b_i) is popped from the stack in part b) of Step 2A, the result follows. □

Lemma 4.2.6 *Consider Step 2A of the algorithm for* $j = k$ *in* G. *If there exists a 1-edge* ik *in* G *with* $i < k$ *such that the triple* (h,a,b) *on the TSTACK has* $a > i$, *then the edge* ik *either crosses the edge* ab *or these two edges are adjacent.*

Proof Since the triple (h,a,b) is on the stack, we must have $b \geq k$ since the j values are considered in decreasing order. Moreover, we must also have $a \leq k$ since otherwise, $j = a$ would have already been considered in Step 2 of the algorithm and, using the result of Lemma 4.2.5, it is easy to see that in Step 2C for $j = a$, the triple (h,a,b)

would have been deleted from TSTACK. If $b = k$, then the two edges ik and ab are adjacent. If $b > k$, then since $a > i$, we conclude that the two 1-edges ik and ab are either adjacent or they cross. □

Lemma 4.2.7 *Consider Step 2B of the algorithm for $j = k$ in G. If the triple (h, a, b) on the stack satisfies highpt$(k) > h$ then the edge ab is either adjacent to edge $(k,$ highpt$(k))$ in G or these two 1-edges cross.*

Proof We must have $h \geq b$. Thus, highpt$(k) > b$. Moreover, since the triple (h, a, b) is on TSTACK, we must have $b \geq k$. We must also have $a \leq k$. Otherwise, $j = a$ would have already been considered in Step 2 of the algorithm and thus, using the result of Lemma 4.2.5, it is easy to see that in Step 2C for $j = a$, the triple (h, a, b) would have been deleted from the stack. If $a = k$ or $b = k$, then the two 1-edges $(k,$ highpt$(k))$ and ab are adjacent. Otherwise, we must have $a < k$ and $b > k$. Thus, we conclude that $a < k < b <$ highpt(k) and that the two 1-edges ab and $(k,$ highpt$(k))$ cross. □

Lemma 4.2.8 *For any connected component $G = (V', E')$ of H', this algorithm can be implemented in $O(|V'|)$ time.*

Proof The number of triples added to TSTACK is $O(|V'|)$ since we only add one triple for every 1-edge in G and since, by construction of the graph H', there are $O(|V'|)$ 1-edges in G. Moreover, each triple may only be deleted if it is on top of the stack. Thus, the time necessary to maintain TSTACK is $O(|V'|)$. Now, consider the graph H constructed by this algorithm. Since, the graph G has $O(|V'|)$ 1-edges, the graph H will also have $O(|V'|)$ nodes. Moreover, edges are only added to H when a triple is deleted from the stack. Since the number of triples added to TSTACK is $O(|V'|)$, the number of edges added to H will also be of this order. Moreover, since any node in G has at most 2 adjacent 1-edges, it is clear that Step 3 of the algorithm can also be implemented in $O(|V'|)$ steps. We can thus construct the graph H in $O(|V'|)$ time and the result follows. □

Lemma 4.2.9 *Let (x, y) be a node in H with h-value equal to h' and suppose that node (x, y) belongs to component C of H. Then there exists some node of the form (x', h') in C.*

Proof It is clear from the way that the h-value is assigned in Step 2A of the algorithm that we either have $h' = y$ or else (x, y) is attached by a path in H to a node (a, b) also with h-value h'. Repeating this argument for the triple (h', a, b) and so on, it is easy to see that (x, y) is attached by a path to some not necessarily distinct node $(x', h') \in C$ with h-value h'. □

Lemma 4.2.10 *If two nodes (a_1, b_1) and (a_2, b_2) in H have the same h-value, then they belong to the same component of H.*

Proof To see this, let h^* be the h-value of nodes (a_1, b_1) and (a_2, b_2). From the result of Lemma 4.2.9, we know that node (a_1, b_1) is in the same component as some not necessarily distinct node (x, h^*). Similarly, (a_2, b_2) is in the same component in H as some not necessarily distinct node (y, h^*). Since edge (x, h^*) is adjacent to edge (y, h^*), we know from Step 3 of the algorithm that the nodes (x, h^*) and (y, h^*) will belong in the same component of H. Hence, the result follows. □

Lemma 4.2.11 *Let C represent a component in H. At any point of the algorithm, if Step 2A has already been considered for $j = h(C)$ and Step 2C has not yet been considered for $j = X(C)$ then there exists a triple of the form $(h(C), x, y)$ on TSTACK. Moreover, we have $(x, y) \in C$.*

Proof To see this, choose a node y^* in G such that $(X(C), y^*) \in C$ and h-value$(X(C), y^*) = h(C)$. Recall that when an edge uv with $u < v$ is considered in Step 2A of the algorithm, its h-value is either equal to v or it is equal to the h-value of another triple on TSTACK. Hence, since we consider the j values in decreasing order in Step 2 and since the h-value of the edge $(X(C), y^*)$ is equal to $h(C)$, at any point

in the algorithm after Step 2A is considered for $j = H(C)$ until edge $(X(C), y^*)$ is considered in Step 2A, there must be a triple on TSTACK with h-value $h(C)$. Now, when edge $(X(C), y^*)$ is considered in Step 2A, the triple $(h(C), X(C), y^*)$ will be added to TSTACK. If this triple is later deleted from TSTACK then it must be in Step 2A, Step 2B or Step 2C of the algorithm. Since by assumption, node $X(C)$ has not yet been considered in Step 2C, the triple must have been deleted either in Step 2A or Step 2B of the algorithm. Now suppose the triple $(h(C), X(C), y^*)$ is deleted in Step 2B. In this case, there will be an edge added in H from some node $(a, \text{highpt}(a))$ to node $(X(C), y^*)$ with $\text{highpt}(a) > h(C)$. Since h-value$(a, \text{highpt}(a)) \geq highpt(a)$, we obtain a contradiction of the definition of $h(C)$. Therefore, if the triple $(h(C), X(C), y^*)$ is deleted from TSTACK, it must be deleted in Step 2A of the algorithm. In this case, the triple $(h(C), X(C), y^*)$ will be replace by some triple of the form (h, a, b) with $a < X(C)$ and $h \geq h(C)$. Moreover, there will be an edge added to join node $(X(C), y^*)$ to node (a, b). Therefore, we have $(a, b) \in C$. If $h > h(C)$ then we obtain a contradiction to the definition of $h(C)$. Moreover, we must have $a \geq X(C)$ since otherwise, we obtain a contradiction of the definition of $X(C)$. Hence, we conclude that the triple $(h(C), x(C), y^*)$ cannot be deleted in Step 2A of the algorithm.

We can therefore conclude that, at any point of the algorithm, if Step 2A has already been considered for $j = h(C)$ and Step 2C has not yet been considered for $j = X(C)$ then there exists a triple of the form $(h(C), x, y)$ on TSTACK. Moreover, from the result of Lemma 4.2.9 and Lemma 4.2.10, it follows that any node in H which has h-value equal to $h(C)$ must belong to component C. □

Lemma 4.2.12 *If we let C represent any component in the graph H obtained using algorithm 4.2.3 on the graph G, then no 1-edge (x, y) in G crosses the edge $(X(C), h(C))$ (Note that this edge does not necessary exist in the graph G). Moreover, there exists no 1-edge in G of the form $(x, X(C))$ with $x < X(C)$ or of the form $(h(C), y)$ with $y > h(C)$.*

Proof Suppose to the contrary that there exists some 1-edge $(x, y), y > x$, in G which crosses the edge $(X(C), h(C))$ or that there exist some 1-edge in G either of the form $(x, X(C))$ with $x < X(C)$ or of the form $(h(C), y)$ with $y > h(C)$. Then, we must either have $x < X(C) \leq y < h(C)$ or $X(C) < x \leq h(C) < y$.

First suppose that we have $x < X(C) \leq y < h(C)$. When edge (x, y) is considered in Step 2A of the algorithm, from the result of Lemma 4.2.11 we know that there will be a triple of the form $(h(C), u, v)$ on TSTACK for some node $(u, v) \in C$ with $u < y$ (or else the triple $(h(C), u, v)$ would have been removed in Step 2C). Moreover, we must have $u \geq X(C)$ since otherwise we contradict the definition of $X(C)$. Since $x < X(C)$ we have $u > x$, and thus, using the result of Lemma 4.2.5, we have that the condition of Step 2Ab) will be true for every triple from the top of TSTACK down to and including the triple $(h(C), u, v)$. Thus we conclude that there will be an edge added from node (u, v) to node (x, y) in H. Moreover the h-value of node (x, y) will be greater or equal to $h(C)$ (again by Lemma 4.2.5). If h-value$(x, y) > h(C)$ then we get a contradiction of the definition of $h(C)$. If h-value$(x, y) = h(C)$ then since $x < X(C)$, we get a contradiction of the definition of $X(C)$. Therefore, we cannot have $x < X(C) \leq y < h(C)$.

Now suppose that $X(C) < x \leq h(C) < y$. In this case, when $j = x$ is considered in Step 2B of the algorithm, from the result of Lemma 4.2.11, we know that there will be a triple of the form $(h(C), u, v)$ on TSTACK with $(u, v) \in C$. Since highpt$(x) \geq y > h(C)$, it follows from the result of Lemma 4.2.5 that in Step 2B there will be an edge added from node $(x, \text{highpt}(x))$ to node (u, v). Therefore, we have $(x, \text{highpt}(x)) \in C$. Since h-value$(x, \text{highpt}(x)) \geq \text{highpt}(x)$, we obtain a contradiction of the definition of $h(C)$.

The result therefore follows. □

Lemma 4.2.13 *Let C represent a component of H at the end of the algorithm. Then we have $X(C) = Y(C)$.*

Proof Since $\{x|(x,y) \in C \text{ and } h\text{-value}(x,y) = h(C)\} \subseteq \{x|(x,y) \in C\}$, it is clear that we must have $Y(C) \leq X(C)$. Now if $Y(C) < X(C)$, then there exists some node of the form $(Y(C), y)$ in C with h-value$(Y(C), y) < h(C)$. Since $(Y(C), y) \in C$, there exists a path in C joining node $(Y(C), y)$ to some node of the form $(X(C), y') \in C$. However by Lemma 4.2.6 and Lemma 4.2.7 we have that the algorithm only joins two nodes in H if they cross or if they are adjacent. Consequently, there must either be a node $(u, v) \in C$ with (u, v) crossing $(X(C), h(C))$ or there must be a node of the form $(x, X(C))$ with $x < X(C)$ and $(x, X(C)) \in C$. This contradicts the results of Lemma 4.2.12. Therefore, we must have $X(C) = Y(C)$. □

Lemma 4.2.14 *If two 1-edges ab and cd in G cross or are adjacent to one other, then the two nodes in H corresponding to these edges belong to the same connected component of H.*

Proof If the 1-edges ab and cd are adjacent then it is clear from Step 3 of the algorithm that they will belong in the same component of H. Therefore, without loss of generality, we can suppose that we have $a < c < b < d$. Let C_1 represent the component of H to which node (a, b) belongs and let C_2 represent the component of H to which node (c, d) belongs. We will show that $C_1 = C_2$.

Before we prove this result in general, we first prove two special cases.

First suppose that $h(C_1) = d$. From the result of Lemma 4.2.9, we know that there exists some node of the form $(x, h(C_1))$ in C_1. Since the 1-edge (c, d) is adjacent to the 1-edge $(x, h(C_1))$, from Step 3 of the algorithm, we know that the two nodes $(x, h(C_1))$ and (c, d) will belong in the same component of H and the result follows.

Now suppose that $X(C_2) = a$. In this case the 1-edge (a, b) will be adjacent to a 1-edge of the form $(X(C_2), y)$ with $(X(C_2), y) \in C_2$. From Step 3 of the algorithm, we know that the two nodes $(X(C_2), y)$ and (a, b) will belong in the same component of H and the result follows.

Now that these two special cases have been treated, consider the more general case.

If $h(C_1) = h(C_2)$ then there exists some nodes $(X(C_1), y_1) \in C_1$ with h-value $h(C_1)$ and $(X(C_2), y_2) \in C_2$ with h-value $h(C_2)$. Since $h(C_1) = h(C_2)$ we conclude from the result of Lemma 4.2.10 that the nodes $(X(C_1), y_1)$ and $(X(C_2), y_2)$ will belong in the same component of H. Therefore we have $C_1 = C_2$ and the result follows.

Now suppose that $h(C_1) < h(C_2)$. From the result of Lemma 4.2.13, we know that $X(C_1) = Y(C_1)$. Moreover, since $a \geq Y(C_1)$, we conclude that $a \geq X(C_1)$. It is clear from the way the h-value is assigned in Step 2A of the algorithm that $h(C_1) \geq b$. If $b \leq h(C_1) < d$ then the 1-edge cd crosses the edge $(X(C_1), h(C_1))$ and we get a contradiction of Lemma 4.2.12. Now suppose $h(C_1) > d$. Since the node (c, d) belongs to C_2 there must be a path in H from node (c, d) to some node of the form $(x, h(C_2))$. However by Lemma 4.2.6 and Lemma 4.2.7 we have that the algorithm only joins two nodes in H if they cross or if they are adjacent. Consequently, since we are assuming $h(C_2) > h(C_1)$, there must either be a node $(u, v) \in C_2$ with (u, v) crossing edge $(X(C_1), h(C_1))$ or else there exists a 1-edge in G of the form $(x, X(C_1))$ with $x < X(C_1)$ or of the form $(h(C_1), y)$ with $y > h(C_1)$. This clearly contradicts the result of Lemma 4.2.12. Therefore, we must have $h(C_1) \geq h(C_2)$.

Now suppose that $h(C_2) < h(C_1)$. From the result of Lemma 4.2.13, we know that $X(C_2) = Y(C_2)$. Moreover, since $c \geq Y(C_2)$, we conclude that $c \geq X(C_2)$. It is clear from the way the h-value is assigned in Step 2A of the algorithm that $h(C_2) \geq d$. If $a < X(C_2) \leq c$ then the 1-edge ab crosses the edge $(X(C_2), h(C_2))$ and we get a contradiction of Lemma 4.2.12. Now suppose that $a > X(C_2)$. In this case, since the node (a, b) belongs to C_1 there must be a path in H from node (a, b) to some node of the form $(x, h(C_1))$. However by Lemma 4.2.6 and Lemma 4.2.7 we have that the algorithm only joins two nodes in H if they cross or if they are adjacent. Consequently, there must either be a node $(u, v) \in C_1$ with (u, v) crossing $(X(C_2), h(C_2))$ or else there exists a 1-edge in G of the form $(x, X(C_2))$ with $x < X(C_2)$ or of the form $(h(C_2), y)$ with $y > h(C_2)$. This clearly contradicts the result of Lemma 4.2.12. Therefore, we

must have $h(C_2) \geq h(C_1)$.

Therefore, we conclude that $h(C_1) = h(C_2)$ and the result follows. □

In the remainder of this chapter, if node (a, b) in H belongs to component K of H, then we will say that the nodes a and b in G belong to component K of H. From the result of the previous lemma, it is clear that every node u in G can belong to at most one component of H. From the construction of the graph H, we know that an edge e in H implies that the two 1-edges in G corresponding to the end-nodes of the edge e are either adjacent or they cross. Therefore, from the results of Lemma 4.2.3 and Corollary 4.2.4, we can conclude that every node in G which belong to the same connected component of H must belong to the same side of any violated cut in G.

Suppose that the graph H obtained using the above algorithm has r connected components and let $T^{(J)}$ be the set of nodes in the graph G which belong in component J of H. Suppose that for each component J, the set $T^{(J)}$ is written as $\{u_1^{(J)}, u_2^{(J)}, \ldots, u_{p_J}^{(J)}\}$ where $u_i^{(J)} \leq u_k^{(J)}$ whenever $i \leq k$. Now, consider the graph G' with associated weight function g', obtained by removing every 1-edge in G and adding, for every $J = 1, \ldots r$ and for every $i = 1, 2, \ldots, p_J - 1$, a 1-edge joining node $u_i^{(J)}$ and node $u_{i+1}^{(J)}$. In Lemma 4.2.15, we show that finding the violations in G is equivalent to finding the violations in G'.

In Figure 4.9, we have drawn the graph G' corresponding to the graph G and H shown in Figure 4.7 and Figure 4.8, respectively.

Lemma 4.2.15 *For any subset of nodes S in G, we have $g(\delta(S)) < 2$ if and only if $g'(\delta(S)) < 2$.*

Proof First suppose that $g(\delta(S)) \geq 2$. If the cut $\delta(S)$ crosses four or more half-edges in G then it is clear that we also have $g'(\delta(S)) \geq 2$. Otherwise, the cut $\delta(S)$ must cross at least one 1-edge in G. Let u and v be the two end-nodes of such a 1-edge. Since every 1-edge in G corresponds to a node in H, the nodes u and v must belong to the

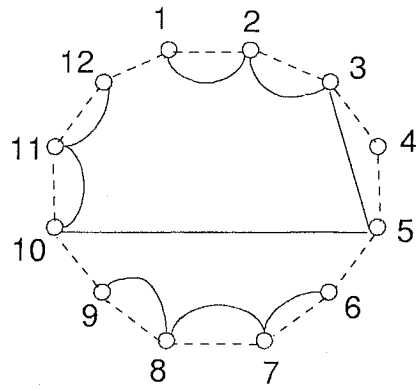

Figure 4.9: The graph G' corresponding to the graph G drawn in Figure 4.7

same component in H. Hence, by construction of the graph G', these two nodes will be joined by a 1-path in G'. Consequently, the cut $\delta(S)$ will have to cross this 1-path in G' and we will have $g'(\delta(S)) \geq 2$.

Conversely, consider the case where $g'(\delta(S)) \geq 2$. Again, if the cut $\delta(S)$ crosses four or more half-edges in G', the result follows. Therefore, suppose that $\delta(S)$ only crosses two half-edges in G'. In such a case, the cut $\delta(S)$ must also cross a 1-edge uv in G'. From the construction of G', we know that nodes u and v must belong to the same side of any violated cut in G. Hence, $g(\delta(S))$ must also have a value greater than or equal to 2. □

Lemma 4.2.16 *No pair of 1-edges in the graph G' obtained using the above transformation cross.*

Proof Suppose to the contrary that there exists two 1-edges ab and cd in G' which cross. Moreover, suppose that nodes a and b belong to component K of H and nodes c and d belong to component J of H. From the construction of the graph G', we must have $J \neq K$. Without loss of generality, suppose that $a < c < b < d$. Let $P_1 = \{u \in V' | u$ belongs to component K of H and $c \leq u \leq d\}$. Since G' has a 1-edge, namely ab, with one end in P_1 and one end out, the same must be true for

G (otherwise, the nodes a and b would not belong to the same component of H). Thus, let xy be a 1-edge in G with one end in P_1 and one end out and suppose that $x < y$. The two nodes x and y must belong to component K of H. Now, let $P_2 = \{u \in V' | u$ belongs in component J of H and $x \leq u \leq y\}$. Again, since G' has a 1-edge, namely cd, with one end in P_2 and one end out, the same must be true for G. Thus, let uv be a 1-edge in G with one end in P_2 and one end out. The nodes u and v must both belong in component J of H. Clearly, the 1-edge uv crosses the 1-edge xy in G. Hence, using the result of Lemma 4.2.14, we must have that $K = J$ and again we obtain a contradiction. □

As we have already mentioned, the graph H will have $O(|V'|)$ nodes and $O(|V'|)$ edges. Thus, finding the connected components of H can be done in $O(|V'|)$ time using depth-first search. It is thus clear that, given any connected component $G = (V', E')$ of H', we can construct the corresponding graph G' in $O(|V'|)$ time.

To find the complete set of cut violations in G', we will use the two shrinking operations defined below, which can be applied anytime G' has 3 or more nodes.

Operation 1. Let u and v be two nodes in G' joined by a half-edge. If there is also an edge of value 1 joining u and v, then any cut which crosses the edge uv must have value greater than or equal to 2 since it must also cross at least one other half-edge in C. Hence, the nodes u and v must belong to the same side of any violated cut constraint in G'. We can therefore contract nodes u and v without modifying the set of violations in G'.

For example, if G' is the graph given in Figure 4.10, then we can shrink nodes u and v without modifying the set of violations. The graph obtained by shrinking nodes u and v is shown in Figure 4.11.

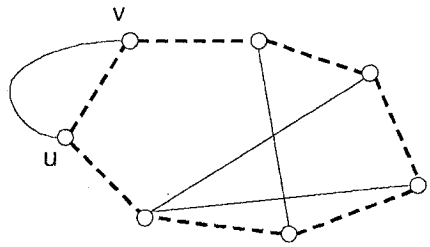

Figure 4.10: The graph G' corresponding to a component of H'

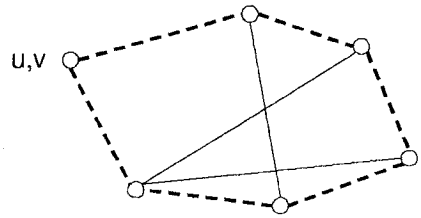

Figure 4.11: Operation 1 on the graph shown in Figure 4.10

Operation 2. Let u be a node in G' with no adjacent 1-edge. Then the value of $g'(\delta(u))$ is 1 and hence, using the result of Lemma 4.2.15, we know that the cut $\delta(u)$ corresponds to a violated cut constraint in G. We therefore need to add this cut to the set of violations of type B found in G. Now let v and w be the two nodes in G' joined to u by a half-edge. Suppose there exists some other cut, $\delta(S)$, in G' of value strictly less than 2 which crosses edge uv or uw. Without loss of generality, we will assume $\delta(S)$ crosses the edge uv. Let xy be the other half-edge in G' crossed by $\delta(S)$. We claim that the cut $\delta(S_1)$, where $S_1 = S \cup \{u\}$, in G' which crosses the edges uw and xy also corresponds to a violation in G'. To see this, note that we have $g'(\delta(S_1)) = g'(\delta(S)) + g'(\delta(u)) - 2g'(u, S)$. Since $g'(u, S) = g'_{uv} = \frac{1}{2}$, we get $g'(\delta(S_1)) = g'(\delta(S)) + 1 - 2(\frac{1}{2}) = g'(\delta(S))$. Hence, the cut $\delta(S_1)$ also corresponds to a violated cut in G'. Using the same arguments, we can also show that every violated cut crossing edge uv corresponds to a violated cut crossing edge uw. Therefore, in order to find the set of violations which crosses edge uv, it suffices to find the set of violations which crosses edge uw. We can therefore safely contract nodes u and

v together. However, we must remember that for every additional violation we find which crosses edge uw, we must also consider the corresponding cut which crosses the edge uv.

An example illustrating this fact is shown in Figure 4.12. Contracting the nodes u and v of the graph drawn in Figure 4.12 results in the graph given in Figure 4.13.

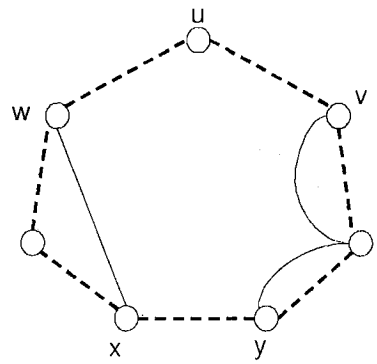

Figure 4.12: The graph G' corresponding to a component of H'

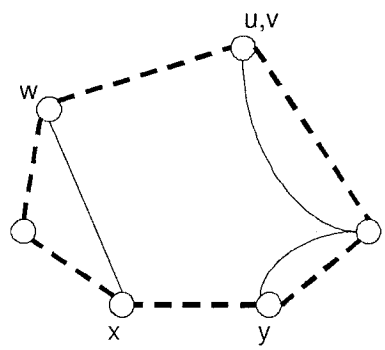

Figure 4.13: Operation 2 on the graph shown in Figure 4.12

Lemma 4.2.17 *Let $\hat{G}$ be any weighted graph on $p \geq 3$ nodes consisting of a half-cycle C and some other 1-edges joining pairs of nodes in the cycle. Moreover, suppose that every node in $\hat{G}$ has at most two adjacent 1-edges and no pair of 1-edges in $\hat{G}$ cross. We can contract $\hat{G}$ to a graph on $p-1$ nodes using one of the two shrinking*

operations described above. Moreover, if $p > 3$ then the resulting graph will have the same properties as $\hat{G}$, i.e., the resulting graph will consist of a half-cycle with possibly some other 1-edges joining nodes of the cycle and every node in this graph will have at most two adjacent 1-edges and no pair of 1-edges will cross.

Proof Suppose to the contrary that we cannot shrink $\hat{G}$ using either of the two shrinking operations outlined above. Since we cannot shrink $\hat{G}$ using Operation 2, every node in $\hat{G}$ must have an adjacent 1-edge. Over all 1-edges in $\hat{G}$, let (x, y) be a 1-edge such that the distance between x and y on C is minimized. Since we cannot shrink using Operation 1, there exists node w which lies on C between x and y. Consider the 1-edge (w, z) adjacent to w. Since no pair of 1-edges cross, we must have that z lies between x and y on C, contradicting the fact that (x, y) was the 1-edge whose endpoints were closest on C.

Moreover, examining these two shrinking operations, we can easily verify that the resulting graph will have the same properties as the graph $\hat{G}$. □

From the previous result and the result of Lemma 4.2.16, it follows that we can repeatedly contract the graph G' corresponding to the graph G until we have a graph with two nodes and, as we will show in Lemma 4.2.18, the following algorithm allows us to do so in $O(|V'|)$ time.

Algorithm 4.2.4 (Algorithm for finding the cut violations in G')

Step 1. (Initialization) Let $i = 1$ and let Nbr_nodes= $|V'|$. For $j = 1, \ldots |V'| - 1$, define next$[j] = j + 1$ and previous$[j + 1] = j$. Moreover, define next$[|V'|] = 1$ and previous$[1] = |V'|$.

Step 2. Let $u =$ previous$[i]$ and $v =$ next$[i]$.

Step 3. If there is a 1-edge joining node i and node v or if node i has no adjacent 1-edge, go to Step 4. Otherwise, go to Step 9.

Step 4. If node i has no adjacent 1-edge then the cut which crosses the half-edges $(i-1 \mod |V'|, i)$ and $(v-1 \mod |V'|, v)$ is a cut of type B in G. Therefore, add the cut $\delta(S)$ where $S = \{i, i+1, \ldots, v-1\}$ to the set of violated cuts of type B in G.

Step 5. Contract node i and v together and re-label the resulting node with label i.

Step 6. Let next[i] be equal to next[v] and let previous[next[v]] be equal to i.

Step 7. If $u \neq |V'|$, let $i = u$.

Step 8. Let Nbr_nodes = Nbr_nodes $-$ 1. If Nbr_nodes > 2, go to Step 2. Otherwise, terminate the algorithm.

Step 9. Let $i = v$ and go to Step 2.

This algorithm considers every node in G' in the order they appear in the half-cycle. For every node i visited, in Step 3 of the algorithm, we verify if nodes i and next[i] can be contracted using either Operation 1 or Operation 2. If these two nodes cannot be contracted, then the algorithm considers the following node next[i]. However, if node i and node next[i] can be contracted, then, if there exists a 1-edge joining node previous[i] and node next[i] then, in the resulting contracted graph, it will be possible to contract node previous[i] and node i using Operation 1. Hence, this is the reason, node previous[i] is revisited in Step 7 of the algorithm.

Lemma 4.2.18 *Given any graph G' corresponding to a component $G = (V', E')$ of H', Algorithm 4.2.4 terminates in $O(|V'|)$ steps.*

Proof Since every node in each of the intermediate graphs has at most two adjacent 1-edges, it is clear that each step in the above algorithm can be implemented in constant time if we use an adjacency list data-structure. Thus, proving this result is equivalent to showing that the number of times Step 8 and Step 9 of the algorithm are executed is

$O(|V'|)$. Since the algorithm terminates when Nbr_nodes is equal to 2, it is clear that Step 8 is executed $|V'| - 2$ times. Moreover, we can easily verify that, at any point during the algorithm, no pair of nodes x and y satisfying $x < i$ and $y < i$ and joined by a half-edge in the current graph can be contracted using Operation 1 or Operation 2. Thus, at any point in the algorithm, if $\hat{G} = (V^*, E^*)$ represents the current graph and we have $i = |V^*|$, then it must be possible to contract node i and node next$[i]$ together using either Operation 1 or Operation 2. Otherwise, we get a contradiction of Lemma 4.2.17. Similarly, if we have $i = |V^*|$ for any intermediate graph $G^* = (V^*, E^*)$, then we conclude that in each additional pass of the algorithm, a contraction will be performed and Step 9 will not be executed. Consequently, this allows us to conclude that Step 9 can be executed at most $O(|V'|)$ times and the result therefore follows. □

When the above algorithm terminates, the number of nodes in the resulting graph is 2. Let u and v be the remaining two nodes in the resulting graph and suppose that $u < v$. If the edges in the resulting graph consist of two half-edges, then the cut $\delta(S)$, where $S = \{u, u+1, \ldots, v-1\}$, corresponds to a violated cut of type B in G, i.e., the cut in G which crosses the half-edges $(u-1, u)$ and $(v-1, v)$ is a cut of type B in G. Otherwise, we have already found the complete set of violations of type B in G.

The following result follows directly from Lemma 4.2.17, Lemma 4.2.18 and the above observation.

Corollary 4.2.19 *Using the two shrinking operations previously described, we can contract the graph G' to a graph with two nodes and hence, find the complete list of violated cuts of type B in $G = (V', E')$. Moreover, Algorithm 4.2.4 allows us to do so in $O(|V'|)$ time.*

Using the result of the above corollary and the fact that the graph G' can be constructed in $O(|V'|)$ time, we conclude that finding the complete list of violations of type B in $G = (V', E')$ can be done in $O(|V'|)$ time.

Now, for each violation $\delta(S)$ found in G, we can obtain the corresponding cut $\delta(S^*)$ in x by taking $S^* = \bigcup_{u \in S}\{\text{Attached\_node}[u]\}$ (see Algorithm 4.2.2).

If the graph corresponding to the support graph of x is connected then the set of cuts of type A and B found using the two algorithms presented in this chapter comprises the complete set of violated cuts of value 1 in x. However, as discussed in the next section, if the graph corresponding to the support graph of x is disconnected then some cuts of value 1 in x still remain to be considered.

For instance, let $\delta(S)$ be a cut of value 1 in x and suppose the support graph of x, denoted by G_x, is disconnected. Suppose the nodes in S belong to component J of G_x and let $V(J)$ represent the node-set of component J. Then the cut $\delta(V(J) \setminus S)$ also has value 1 in x and thus also corresponds to a violated cut constraint in x. Consequently, if G_x is disconnected, for every cut $\delta(S)$ of value 1 found in x, we will also have to consider the violated cut $\delta(V(J) \setminus S)$.

In Figure 4.14 and Figure 4.15, we show G_z for a point z in Q_F^{12} and G_x for a vertex $x \in Q_F^{12}$ present in the convex combination of z. The cut $\delta(S)$ where S is equal to the node-set $\{5, 6, 7, 8\}$ is a cut of value 1 in x. If we compute $z(\delta(S))$, we get a value of 3. Hence the cut $\delta(S)$ does not correspond to a violated cut in z. However, as can be verified in Figure 4.14, the cut $\delta(V(J) \setminus S)$, where $V(J) = \{1, 2, 3, 4, 5, 6, 7, 8\}$, is violated by z.

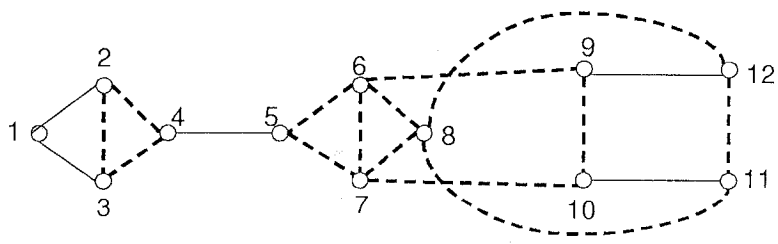

Figure 4.14: The graph G_z for a point z in Q_F^{12}

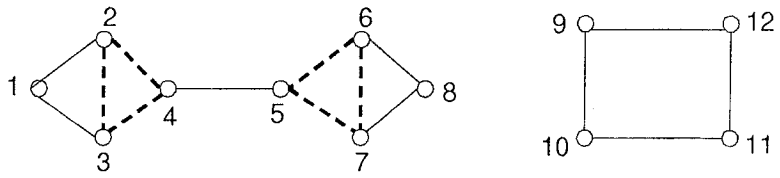

Figure 4.15: The graph G_x for a vertex x of Q_F^{12}

Moreover, if G_x is disconnected, then any set of connected components of G_x can be added to the nodes in S to form another cut of the same value. The algorithm we are constructing will not consider the cuts formed by adding every set of connected components. Remember that for every violation found in x, we have to verify the value of the associated cut constraint in z and, if G_x is disconnected, the number of violations in x will be exponential in the number of components. Since we cannot possibly consider an exponential number of violations, this restriction is reasonable. However, if the only cut constraint violated in z is a constraint obtained by adjoining a set of connected components of G_x to the nodes in S, then this cut might remain hidden even after we consider every vertex in the convex combination of z.

For example, consider the point $z \in Q_F^{14}$ whose support graph is shown in Figure 4.16 and let the vertex x of Q_F^{14} corresponding to the graph shown in Figure 4.17 be a vertex present in the decomposition of z. The cut $\delta(S)$ where S is the node-set $\{4, 5, 6, 7\}$ is a violated cut in x. The value of $z(\delta(S))$ is 3 and hence $\delta(S)$ does not correspond to a violated cut in z. We can also verify that the cut $\delta(T \setminus S)$ where T is the node-set $\{4, 5, 6, 7, 8, 9, 10, 11\}$ is also satisfied by z. However, if we let S_2 be the node-set $\{1, 2, 3\}$ then the constraint corresponding to $\delta(S \cup S_2)$ is violated by z. If we do not consider the cuts formed by adjoining to S the connected components of G_x, then this violated cut might remain hidden even after we consider every vertex in the convex combination of z.

To resolve this problem, let $S_1, S_2, \ldots, S_p$ be the node-sets of the connected components of G_x and suppose the nodes in S all belong to S_1. Consider the complete

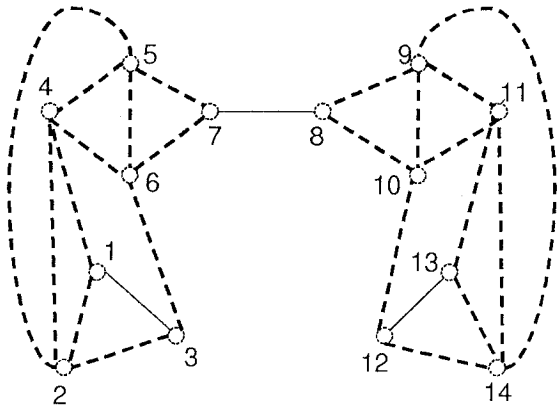

Figure 4.16: The graph G_z for a point z in Q_F^{14}

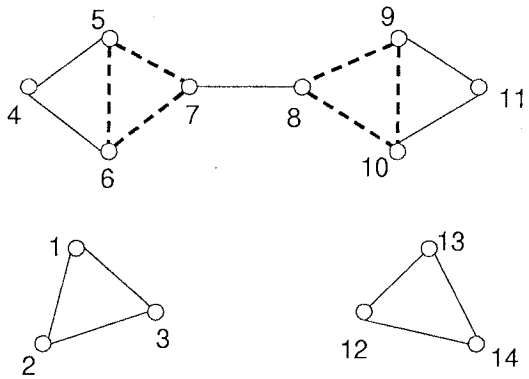

Figure 4.17: A graph G_x for a vertex x of Q_F^{14}

weighted graph, K_{p+1}, defined as follows: $K_{p+1} = G_z \circ S \circ (S_1 \setminus S) \circ S_2 \circ S_3 \circ \ldots \circ S_p$. Note that if the process of shrinking the graph G_z creates multiple edges, then we simply merge then and add their weight values. Let u and u' be the two nodes in K_{p+1} representing the node-sets S and $S_1 \setminus S$, respectively. It is clear that if a violated cut constraint in z can be found by adjoining to the nodes in S a set of connected components of G_x, then the minimum uu'-cut in K_{p+1} will have a value stricly less than 2. In fact, the nodes in K_{p+1} which are on the same side of the minimum cut as u represent the set of connected components of G_x which can be added to S to form a violated subtour constraint in z. On the other hand, if the minimum uu'-cut in K_{p+1}

has value greater than or equal to 2, then no set of connected components of G_x can be added to S to form a violation in z.

Thus, whenever the support graph of x has more than two connected components, for every violated cut $\delta(S)$ of value 1 found in x, we will construct the corresponding graph K_{p+1} and use any fast minimum st-cut code to find the minimum uu'-cut in K_{p+1}. Thus, if it is possible to find at least one violation in z when we adjoin to S any set of connected components of G_x, then by doing so, we are guaranteed to find one of these violations. There exists many polynomial-time algorithms for computing minimum st-cut. For example, the push-relabel algorithm introduced by Goldberg and Tarjan in 1988 can be implemented in $O(nm\log(n^2/m))$ steps on a graph with m edges and n nodes [CG1994]. Although, in theory, the number of components in a vertex of Q_F^n can be as large as $\frac{n}{3}$, as we will show in Chapter 6, in all the vertices with which we worked while testing this separation algorithm, the largest number of components observed was 56. Hence, in practice, the number of connected components in a vertex of the fractional 2-factor polytope is relatively small compared to the number of nodes in the vertex and thus the time needed to find the minimum uu'-cut in K_{p+1} is, in most cases, negligible.

Even though the number of components in the vertices of Q_F^n is, in practice, relatively small, one might still argue that constructing the graph K_{p+1} and computing the minimum uu'-cut in K_{p+1} for every cut of value 1 found in x will be very expensive and that the number of additional cuts in z found by doing this supplementary work will not justify the extra effort invested. However, as will be show in Chapter 6, we found that, in most examples, incorporating this extra step surprisingly did reduce the overall running time of the cutting plane process.

Now, consider again the point z and the vertex x whose support graphs are shown in Figure 4.16 and 4.17, respectively, and let S be equal to the node-set $\{4,5,6,7\}$. To verify if the nodes in S can be added to any set of connected components of G_x to

form a violated cut in z, it suffices to construct the corresponding graph K_{p+1} recently described. Since G_x has three components, we have $p = 3$. Moreover, if we let S_1 be the set of nodes in the connected component of G_x which contains the nodes in S, then we have $S_1 = \{4, 5, 6, 7, 8, 9, 10, 11\}$. If we let S_2 be the node-set $\{1, 2, 3\}$ and S_3 be the node-set $\{12, 13, 14\}$, then we have $K_{3+1} = G_z \circ S \circ (S_1 \setminus S) \circ S_2 \circ S_3$. This graph is illustrated in Figure 4.18. Recall that the nodes u and u' in this graph represent the node-sets S and $S_1 \setminus S$, respectively. By computing the minimum uu'-cut in this graph, we find a cut of value 1 given by the node u and the node representing the set S_2. Therefore the set of nodes in S_2 can be ajoined to the set of nodes in S to form a violated cut of value 1 in z.

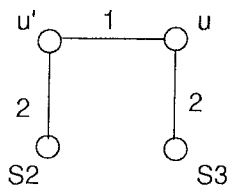

Figure 4.18: The graph K_4 obtained by contracting the graph shown in Figure 4.16

Now, as discussed in Chapter 1, to find the complete set of violated subtour constraints in z, it suffices to consider the complete set of violations in each of the vertices present in the decomposition of z. Since every violation present in a vertex x of the fractional 2-factor polytope must correspond to either:

i) a single connected component of the support graph of x, denoted by G_x,

ii) a set of connected components of G_x,

iii) a cut of value 1 where every node in the cut belongs to a single connected component of G_x, or

iv) a cut of type iii) added to a set a connected components of G_x,

we know that by considering every such violation, we will find the complete set of cut violations in z.

Using the algorithm described in this chapter, it is clear that every cut of type i) and iii) in x will be considered. Moreover, if at least one cut of type i) or ii) is violated in z, then we are guaranteed to find one of these cuts when we compute the minimum cut in the complete graph, K_p, described in Section 4.1. Similarly, for every cut, $\delta(S)$, of type iii) found in x, if either S or S added to a set of connected components of G_x, correspond to a violated cut constraint in z, then we are sure to find such a violation when we compute the minimum uu'-cut in the complete graph K_{p+1} described earlier.

Thus, given any point z in Q_F^n, we are guaranteed to find at least one violation in z if one exists. Hence, the algorithm constructed in this chapter is an exact separation algorithm.

Chapter 5

Computer Implementation and Enhancements

In this chapter we give a detailed description of our implementation of a cutting plane algorithm for the Subtour Elimination Problem. This cutting plane framework makes use of the algorithms described in the previous chapters to find violated subtour constraints. The program is called CONVCOMB-SEP and contains roughly 5000 lines of code. This code was written in ANSI C and compiled using the GNU C compiler (Version 2.7.2).

Several enhancements that we made to speed up the cutting plane algorithm are also discussed in this chapter. Moreover, we explain how the cutting plane framework had to be adapted to solve large-scaled Subtour Elimination Problems. Several of the enhancements presented in this chapter have been taken from the literature and will be referenced accordingly.

At the end of this chapter, we give a detailed pseudo-code description of the cutting plane algorithm implemented.

5.1 Heuristics to find violated cut constraints

Let z be a point in Q_F^n. Notice that if the support graph corresponding to the vector z is disconnected then it is not necessary to apply the separation algorithm described in Chapter 1 to find violated subtour constraints in z. If the support graph of z, denoted by G_z, is disconnected, then each connected component of G_z defines a violated constraint in z. In fact, any set of connected components defines a violated constraint, giving a number of violated constraints exponential in the number of components. In our implementation of CONVCOMB-SEP, we only consider the constraints defined by one component. This choice makes sense because, if each connected component is forced to join with another one, then we at least halve the number of components and hence we make good progress.

In the paragraphs which follow, we present another heuristic used to find violated cut constraints. This heuristic is designed to work with points in $\mathbb{R}^E$ which have a connected support graph. Nonetheless, if the support graph corresponding to a point z in Q_F^n is disconnected, then in addition to considering the connected component cuts described above, we can also use the heuristic presented below to find additional violated cut constraints in each of the connected components of G_z. This heuristic has previously been introduced in [B1986].

In this heuristic procedure, given a point $z \in \mathbb{R}^E$, we first find a maximum cost spanning tree in K_n with respect to the edge cost z_e for all edges e in E. Such a spanning tree can be found in $O(|E| + |V| \log |V|)$ steps using Fibonacci heaps (see [FT1987]). Now, if the support graph of z is connected, then to compute such a spanning tree it suffices to consider the edges which have $z_e > 0$. From our experiments, we found that, in practice, the number of edges in the support graph of any point z obtained during the cutting plane process is of order $|V|$. Hence, in practice, such a maximum spanning tree can be computed in $O(|V| \log |V|)$ steps. Then, letting $S_{\max}$ be the set of all $S \subseteq V, 3 \leq |S| \leq n-3$ such that S is one side of a node partition obtained

by removing some edge from the maximum spanning tree, we check all the subtour constraints corresponding to every set S in $S_{\max}$ for feasibility in z. This method is not guaranteed to find an existing violated subtour constraint. However, as it is shown in [B1986], we do have the following result.

Theorem 5.1.1 *Let $z \in \mathbb{R}^E$ satisfy the constraints of the Fractional 2-Factor Problem, let G_z represent the support graph of z, and let T be a maximum cost spanning tree of $K_n = (V, E)$ with respect to the costs z_e for all $e \in E$. Then if $z(\delta(S)) \geq 2$ for every $S \subseteq V, 3 \leq |S| \leq n-3$ such that S is one side of a node partition obtained by removing some edge from T, it follows that G_z is 2-edge connected.*

We found this algorithm to be a very efficient heuristic for finding violated cut constraints. Hence we decided to make use of it in our cutting plane algorithm.

Note that if G_z is disconnected then we can use a separate processor to execute this heuristic on each connected component of G_z in parallel. Moreover, for each cut corresponding to a set S in $S_{\max}$, we can use a separate processor to verify the value of this cut in z.

Although both of these heuristics can be implemented very efficiently, we found that, in practice, it is better not to wait until both of these heuristics fail to find a violated cut constraint in z before making use of the decomposition algorithm described in Chapter 1. From our experiments, we found that it was more profitable to run the decomposition algorithm described in Chapter 1 together with these two heuristics as soon as the support graph of z became connected. Thus, in our implementation of CONVCOMB-SEP, given any point z in Q_F^n, we first verify if z belongs to Q_S^n by running any fast minimum cut algorithm on the support graph corresponding to the vector z, denoted by G_z. If the value of the minimum cut in G_z is greater than or equal to 2, then we are done and there is no need to search for violations in z. Otherwise, we start by verifying the connectivity of the graph G_z. If this graph is disconnected then every connected component of G_z defines a violated subtour constraint in z. We thus

add these violations to our current LP. Then, for every connected component of G_z, we run the heuristic described above to find additional violations is z. Every additional violation found is added to the LP. Moreoever, if G_z is connected then we proceed to use the algorithm described in Chapter 1 to find violations in z. The additional violations found are added to the current LP. At the end of this process, we re-solve the LP obtained and we carry on with the cutting plane process.

5.2 Removing inactive cut constraints

At every step of the optimization process, the amount of time required by the LP solver to solve the current LP is greatly affected by the size of the LP we are optimizing. Hence, the number of subtour constraints included in a given LP has a significant influence on the amount of time required to solve this LP. Consequently, if a cut constraint is not playing an active role in the LP, i.e., if it has a value stricly greater than 2, then it may be desirable to remove it so that the LP solver does not have to deal with it in the future. This enhancement is suggested in [L2000].

As will be shown in Chapter 6 when we discuss the computational results of CONVCOMB-SEP, our experiments have demonstrated that, effectively, eliminating inactive cut constraints did reduce the running time of CONVCOMB-SEP. We therefore decided that, at every step of the optimization process of the newly added constraints, we would only keep the ones which were in the basis. In other words, a constraint only stays in the LP if it plays an active role in the optimum when it is added. Moreover, for simplicity reasons, we assume that if a constraint is kept once, it is sufficiently relevant to stay in until the end of the process.

5.3 Contracting out paths consisting of edges of weight one

In our implementation of CONVCOMB-SEP, we also use the following heuristic to speed up our algorithm for finding violated cut constraints.

Given a point z in Q^n_F, before we start looking for violations in z, we first contract the paths consisting of edges of weight one in the support graph of z to single 1-edges. Denote by $\tilde{G}_z$ the graph obtained by shrinking the 1-paths in the support graph of z (i.e., the paths which consist of edges of weight one) and denote by $\tilde{z}$ the vector corresponding to the graph $\tilde{G}_z$. Note that this process might create parallel edges in $\tilde{G}_z$. For example, consider the point z whose support graph is shown in Figure 5.1. If we contract the paths consisting of edges of weight one in the support graph of z to single 1-edges, then the graph we obtain is the graph illustrated in Figure 5.2.

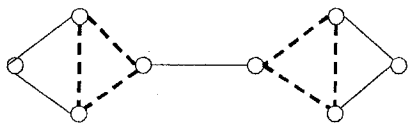

Figure 5.1: The support graph of a point in Q^8_F

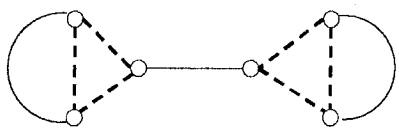

Figure 5.2: Contracting 1-paths in the graph shown in Figure 5.1

The point of this heuristic is that if any edge on a path consisting of edges of weigth one is in a cut violation, then every such edge will also be in a cut violation, so we will find many violations which are very similar. From our experiments and from the results suggested in [L2000], we found that it was more useful to find violated cut constraints which were different. So we decided to use this heuristic to avoid finding too many similar cuts.

As we will show in Chapter 6, in most practical examples, this heuristic also allows us to considerably reduce the size of the graphs we are working with and, as we will also demonstrate in Chapter 6, this turns out to have a noticeable effect on the total time required by the cutting plane algorithm. In the following paragraph, we give some indications as to why this is true.

If we implement the enhancements described in Sections 3.2.1 and 3.2.2 of Chapter 3, then it is clear that the amount of work required to find a convex decomposition for the vector $\tilde{z}$ is equivalent to the amount of work required to find a convex decomposition for our original vector z. However, if the dimension of $\tilde{z}$ is smaller than the dimension of z, then the dimension of the vertices in the convex combination of $\tilde{z}$ will also be smaller than the dimension of the vertices present in the convex combination of z. Hence, using the algorithm described in Chapter 4, it will be faster to find the list of violations in the vertices present in the decomposition of $\tilde{z}$ than in the vertices present in the decomposition of z. Moreover, as explained in Chapter 1, every violation found in the vertices present in the convex combination of z has to be checked in z. It is easy to see that verifying a constraint in z is equivalent to verifying the corresponding constraint in $\tilde{z}$. Since in most cases, the dimension of the vector $\tilde{z}$ is much smaller than the dimension of z, it will be faster to verify the cut constraint in $\tilde{z}$.

Hence, if the dimension of $\tilde{z}$ is smaller than the dimension of z, then shrinking the 1-paths in the support graph corresponding to the vector z allows us to speed up the separation algorithm described in Chapter 1. Since the dimension of $\tilde{z}$ is smaller than the dimension of z in most practical examples (see Chapter 6), we have decided to implement this heuristic in our program CONVCOMB-SEP.

5.4 Working with a small portion of the edge-set

In this section, we describe how CONVCOMB-SEP had to be adapted to solve large-scaled Subtour Elimination Problems.

5.4.1 Storage constraint

At every step of the cutting plane process the size of the LPs we are optimizing is very big. Recall from Chapter 1 that the starting system of constraints used for the cutting plane approach for the Subtour Elimination Problem is given by system 1.11. Hence, even in the first step of the cutting plane process, the number of variables in the LP we are solving will be of order n^2. Moreover, as the number of degree constraints alone is n, storing the entire constraint matrix for a problem with, eg., 1000 or more nodes explicitly would not be practical or even possible with the present computer technology.

On the other hand, even if it would be possible to store the entire constraint matrix, we have to take into account that the time taken by the LP solver to optimize a problem is strongly affected by the size of the LP, and consequently, by the number of variables in the LP program. Hence, even if storage limitation is not an issue, working with the complete set of variables is disadvantageous.

In every vertex of Q_S^n, no more than $2|V|-3$ out of the $n(n-1)/2$ variables will be non-zero (see [BL2002]). Since every Subtour Elimination Problem will be optimized at a vertex of Q_S^n, this observation gives us hope that we might be able to overcome this problem by working with only a small portion of the variables at each step of the cutting plane process. The variables which are not included in a given LP are simply assumed to have a value of 0. However, note that if we do not work with the complete set of variables, then every time we find an optimal solution which satisfies all the subtour elimination constraints, i.e., a solution which has a minimum cut of value greater than or equal to 2, we will need to verify the remaining set of variables to see if any of them might be needed, i.e., might be non-zero in an optimal solution. In Section 5.4.4, we will illustrate how the dual of the current LP can be used to verify if a given variable needs to be included in the current LP.

5.4.2 Choosing an initial edge-set

In this section, we describe an efficient procedure implemented in CONVCOMB-SEP which allows us to obtain a substantial reduction in the number of variables.

In this section, we define the k-nearest neighbour subgraph of a graph K_n to be the subgraph of K_n which contains for every node in K_n all edges to its k nearest neighbours.

In previous experiments (see [JRR1994]), it had been observed that a large portion of the non-zero edges in an optimal subtour solution connect near neighbours. Moreover, the experiments conducted have shown that in almost all instances, the optimal solutions are contained in the 50-nearest neighbour subgraph of K_n, and that, in every example, a significant portion of the edges in an optimal solution are contained in the 7-nearest neighbour subgraph of K_n.

As these observations suggest, the edges contained in the 7-nearest neighbour subgraph of K_n are good candidate edges for our initial edge-set. In addition, notice that if our initial edge-set is not Hamiltonian, then at some point in the cutting plane process, the LP we will be trying to solve will be infeasible. To resolve this problem, we augment the 7-nearest neighbour subgraph of K_n by the edges of a Hamiltonian cycle found using any simple heuristic. In our implementation, we used the MST-based 2-approximation algorithm for TSP (see [CLRS2001]) to compute a Hamiltonian cycle in the complete graph K_n. The graph obtained by augmenting the 7-nearest neighbour subgraph of K_n with this Hamiltonian cycle is referred to as the *sparse graph*. Moreover, since we found that, in practice, the set of edges in the minimum spanning tree were also good candidate edges, we decided to include those edges in the sparse graph as well. At any point in the cutting plane process, we refer to the edges in the sparse graph as the set of active edges and we refer to the other edges as non-active.

As is also suggested by the above observations, of the remaining edges, some edges are more likely than others to be non-zero in an optimal solution. More precisely, as

we have previously stated, in almost every practical example, the optimal solution is contained in the 50-nearest neighbour subgraph of K_n. Hence, of the non-active edges, the edges in this subgraph are more prone to be non-zero in an optimal solution. To make use of this fact, we will compute a list of edges which have to be added to the initial set of active edges to obtain the 50-nearest neighbour subgraph of K_n. These edges will form the *reserve graph*. Furthermore, whenever we need to verify the set of non-active edges to see if any of these edges needs to be added to the set of active edges, we will first consider the edges in this reserve graph. Only in situations where no edges in the reserve graph have been added to the set of active edges will we carry on to verify the remaining non-active edges. Verifying the complete set of non-active edges can be very time consumming. Hence, by computing a reserve graph, we considerably diminish the running time of CONVCOMB-SEP.

Since computing a minimum spanning tree of the complete weighted graph K_n can be implemented in $O(n^2)$ time (see [PS1982]), computing a Hamiltonian cycle using the MST-based 2-approximation algorithm for TSP can also be implemented in $O(n^2)$ time. Moreover, it is easy to see that finding the 7-nearest and the 50-nearest neighbour subgraph of K_n can also be implemented in $O(n^2)$ time. Thus, constructing the initial sparse graph and the initial reserve graph can be implemented in $O(n^2)$ time. However, if we assume that we are working in a parallel environment with p processors then it is clear that both the 7-nearest and the 50-nearest neighbour subgraphs of K_n can be constructed in $O(n^2/p)$ time. Thus, if we have $O(n)$ processors then these two subgraphs can both be implemented in $O(n)$ time.

5.4.3 When to verify the set of non-active edges

As previously mentioned, whenever the current optimal solution satisfies all the subtour constraints, we will need to go through the list of non-active edges and verify if any of them needs to be added to the set of active edges, i.e., if any of the non-active

edges might be non-zero in an optimal solution. Considering the observations made in Section 5.4.2, whenever we need to verify the non-active edges, we will first start by considering the edges in the reserve graph. If at least one of these edges needs to be added to our current active set of edges, then we will add the variables corresponding to these edges to our current LP, re-solve the LP obtained and carry-on with the cutting plane process. Otherwise, the remaining non-active edges will be verified and as long as one non-active edge is added to the set of active edges, the cutting plane process will continue. However, if none of the remaining non-active edges needs to be added to the set of active edges, then we are guaranteed that our current optimal corresponds to an optimal solution when we consider every edge of the complete graph K_n.

Although the correctness of the algorithm does not require this, we will also verify the edges in the reserve graph after every fifth iteration of the cutting plane process. Since the non-active edges which are required in a good or optimal subtour tend to be added to the set of active edges early in the computations, by increasing the frequency of the edge verification, we can usually achieve a valid optimal solution faster. Moreover, in [L2000], it is observed that when the set of non-active edges is not verified frequently enough, then the set of active edges seems to grow too much and this actually slows down the LP solver. On the other hand, checking the set of edges in the reserve graph is time consuming. Hence, we cannot verify the reserve graph after every iteration of the cutting plane process. From our experiments, we found a frequency of one edge check every fifth iteration to be a good compromise. The experiments done in [L2000] and [PR1991] also confirm this observation.

5.4.4 Verifying the set of non-active edges

At every step of the cutting plane process, the set of non-active edges are assumed to be zero in an optimal solution. Consequently, whenever we find an optimal solution which satisfies every subtour elimination constraint, we must verify that this optimal

solution also corresponds to a global optimal solution, i.e., that every non-active edge is effectively zero in an optimal solution which considers every edge of the complete graph K_n. In this section we describe an algorithm which can be used to verify if a given non-active edge needs to be included in the current set of active edges.

Given a set $J \subseteq E$, a vector $x \in \mathbb{R}^E$ and a matrix B whose columns are indexed by the elements of E, we denote by x_J the projection of x into the subspace $\mathbb{R}^J$ and by B_J, the submatrix of B consisting of the columns with index in J.

If we let J represent the current set of active edges and let $\mathcal{L}$ represent the set of violated cuts found so far, then the current LP can be written as follows:

$$\begin{aligned} \text{minimize } & c_J x_J \\ \text{subject to: } & A_J x_J = 2 \\ & L_J x_J \geq 2 \\ & 0 \leq x_J \leq 1 \end{aligned} \tag{5.1}$$

where $A \in \mathbb{R}^{V \times E}$ is the node-edge incidence matrix of K_n and L is the matrix representation of the set $\mathcal{L}$.

The dual corresponding to LP 5.1 has variables $y \in \mathbb{R}^V$, $u \in \mathbb{R}^E$ and $d \in \mathbb{R}^{\mathcal{L}}$ and is defined as follows:

$$\begin{aligned} \text{maximize } & 2y - u + 2d \\ \text{subject to: } & y_i + y_j - u_{ij} + \sum(d_S : S \in \mathcal{L}, ij \in \delta(S)) \leq c_{ij} \quad \text{for all edges } ij \in J \\ & u_e \geq 0 \quad \text{for all edges } e \in J \\ & d_S \geq 0 \quad \text{for all } S \in \mathcal{L}. \end{aligned} \tag{5.2}$$

Hence if $\overline{x}_J$ is an optimal solution to LP 5.1, then by duality and complementary slackness theory, we know that there exists a corresponding feasible solution $(\overline{y}, \overline{u}, \overline{d})$ to LP 5.2 which also satisfies the following equalities:

$$\begin{aligned} \overline{y}_i + \overline{y}_j - \overline{u}_{ij} + \sum(\overline{d}_S : S \in \mathcal{L}, ij \in \delta(S)) = c_{ij} \quad & \text{for all } ij \in J, \overline{x}_{ij} > 0 \\ \overline{u}_e = 0 \quad & \text{for all } e \in J, \overline{x}_e < 1 \\ \overline{d}_S = 0 \quad & \text{for all } S \in \mathcal{L}, L_S \overline{x}_J > 2. \end{aligned}$$

Now consider the vector $\overline{x}^* \in \mathbb{R}^E$ corresponding to the solution $\overline{x}_J$, i.e.

$$\overline{x}^*_e = \begin{cases} \overline{x}_e & \text{if } e \in J \\ 0 & \text{otherwise.} \end{cases}$$

If every non-active variable is indeed zero in an optimal solution when we consider every edge in the complete graph K_n, then the vector $\overline{x}^*$ will be an optimal solution to the LP defined below.

$$\begin{aligned} \text{minimize } & cx \\ \text{subject to: } & Ax = 2 \\ & Lx \geq 2 \\ & 0 \leq x_J \leq 1. \end{aligned} \tag{5.3}$$

Again, it follows from the theory of duality and complementary slackness that $\overline{x}^*$ is an optimal solution to LP 5.3 if and only if there exists a corresponding feasible solution $(\overline{y}^*, \overline{u}^*, \overline{d}^*)$ to the following set of constraints:

$$\overline{y}^*_i + \overline{y}^*_j - \overline{u}^*_{ij} + \sum(\overline{d}^*_S : S \in \mathcal{L}, ij \in \delta(S)) \leq c_{ij} \quad \text{for all } ij \in E \tag{5.4}$$

$$\overline{u}^*_{ij} \geq 0 \quad \text{for all } ij \in E \tag{5.5}$$

$$\overline{d}^*_S \geq 0 \quad \text{for all } S \in \mathcal{L} \tag{5.6}$$

which also satisfies the following complementary slackness conditions:

$$\overline{y}^*_i + \overline{y}^*_j - \overline{u}^*_{ij} + \sum(\overline{d}^*_S : S \in \mathcal{L}, ij \in \delta(S)) = c_{ij} \quad \text{for all } ij \in E, \overline{x}^*_{ij} > 0 \tag{5.7}$$

$$\overline{u}^*_{ij} = 0 \quad \text{for all } ij \in E, \overline{x}^*_e < 1 \tag{5.8}$$

$$\overline{d}^*_S = 0 \quad \text{for all } S \in \mathcal{L}, \overline{x}^*(\delta(S)) > 2. \tag{5.9}$$

Hence, if $\overline{x}^*$ is an optimal solution to LP 5.3, then we must have $\overline{u}^*_e = 0$ for every edge e not in J (this follows from condition 5.8 above). Moreover, any feasible solution, $(\overline{y}^*, \overline{u}^*, \overline{d}^*)$, satisfying constraints 5.4 through 5.9 will have to satisfy $\overline{y}^*_i = \overline{y}_i$ for all $i \in \{1, 2, \ldots, n\}$, $\overline{d}^*_S = \overline{d}_S$ for all $S \in \mathcal{L}$ and $\overline{u}^*_e = \overline{u}_e$ for every edge e in J. Hence, if $\overline{x}^*$ is an optimal solution to LP 5.3, then we must have

$$(\overline{y}^*, \overline{u}^*, \overline{d}^*) = (\overline{y}, \tilde{u}, \overline{d}) \quad \text{where } \tilde{u}_e = \begin{cases} \overline{u}_e & \text{if } e \in J, \\ 0 & \text{otherwise.} \end{cases} \tag{5.10}$$

The solution $(\overline{y}^*, \overline{u}^*, \overline{d}^*)$ given by equation 5.10 clearly satisfies conditions 5.5 throught 5.9. Moreover, it is clear that condition 5.4 is verified for every edge $ij \in J$. Therefore, in order to verify if any of the non-active edges might be non-zero in an optimal solution, it remains to verify that the following constraint 5.11 is satisfied for every edge ij not belonging to J:

$$\overline{y}_i + \overline{y}_j + \sum(\overline{d}_S : S \in \mathcal{L}, ij \in \delta(S)) \leq c_{ij} \tag{5.11}$$

Any edge ij which does not satisfy the previous constraint might be non-zero in an optimal solution, and hence, must be added to the set of active edges J. On the other hand, if every edge ij not belonging to J satisfies condition 5.11, then we know that the solution $\overline{x}^*$ corresponds to an optimal solution to LP 5.3.

Verifying constraint 5.11 for a given edge ij is equivalent to verifying that the *reduced cost* $r_{ij} = c_{ij} - \overline{y}_i - \overline{y}_j - \sum(\overline{d}_S : S \in \mathcal{L}, ij \in \delta(S))$ is greater than or equal to zero.

Computing the reduced cost of an edge e

To speed up the computation of the reduced cost r_{ij} of an edge ij, we exploit the fact that the vector $\overline{d} \in \mathbb{R}^{\mathcal{L}}$ is non-negative. Hence, if we initialize r_{ij} to be equal to

$c_{ij} - \overline{y}_i - \overline{y}_j$, then we can stop the computation of $r_{ij} - \sum(\overline{d}_S : S \in \mathcal{L}, ij \in \delta(S))$ as soon as the partial result becomes negative, indicating that the edge ij must be added to the set of active edges. However, since only a small portion of the non-active edges will have a negative reduced cost, in most cases, this modification will have no effect on the computation time. Nonetheless, note that r_{ij} is also equal to $c_{ij} - \overline{y}_i - \overline{y}_j - \sum(\overline{d}_S : S \in \mathcal{L}) + \sum(\overline{d}_S : S \in \mathcal{L}, ij \notin \delta(S))$. Hence, if we initialize r_{ij} to be equal to $c_{ij} - \overline{y}_i - \overline{y}_j - \sum(\overline{d}_S : S \in \mathcal{L})$, then we can stop the computation of $r_{ij} + \sum(\overline{d}_S : S \in \mathcal{L}, ij \notin \delta(S))$ as soon as the partial summation becomes positive, indicating that condition 5.11 is satisfied for the edge ij. Since most edges have a positive reduced cost, in our experimental computations, we found that this modification allowed us to cut the computation time by a factor of at least two.

To reduce the computation time even further, notice that we can also write r_{ij} as $c_{ij} - \overline{y}_i - \overline{y}_j - \sum(\overline{d}_S : S \in \mathcal{L}, i \in S) - \sum(\overline{d}_S : S \in \mathcal{L}, j \in S) + \sum(\overline{d}_S : S \in \mathcal{L}, ij \in \gamma(S))$. Since in most cases, $\sum(\overline{d}_S : S \in \mathcal{L}, i \in S) + \sum(\overline{d}_S : S \in \mathcal{L}, j \in S)$ will be much smaller than $\sum(\overline{d}_S : S \in \mathcal{L})$, initializing r_{ij} to be equal to $c_{ij} - \overline{y}_i - \overline{y}_j - \sum(\overline{d}_S : S \in \mathcal{L}, i \in S) - \sum(\overline{d}_S : S \in \mathcal{L}, j \in S)$ and stopping the computation as soon as the partial sum of $r_{ij} + \sum(\overline{d}_S : S \in \mathcal{L}, ij \in \gamma(S))$ becomes positive allowed a significant improvement on the computation time required in each of the examples that we looked at.

5.4.5 Pseudo-code of the edge generation process

Here is a pseudo-code description of the edge generation process:

Build an initial sparse graph consisting of the 7-nearest neighbour subgraph of K_n and the edges of any greedy tour of K_n

Build a reserve graph consisting of a set of edges which have to be added to the initial sparse graph to form the 50-nearest neighbour subgraph of K_n

repeat

repeat

add cuts

if no cuts added or a fifth pass through loop

verify edges in reserve graph to see if any of them need to be added

until no cuts added and no edges added

check remaining edges to see if they need to be added

until no edges added

5.5 Pseudo-Code of CONVCOMB-SEP

In this section, we give a full pseudo-code description of CONVCOMB-SEP.

Build an initial sparse graph consisting of the 7-nearest neighbour subgraph of K_n and the edges of any greedy tour of K_n

Build a reserve graph consisting of a set of edges which have to be added to the initial sparse graph to form the 50-nearest neighbour subgraph of K_n

Initialize LP to

$$\begin{aligned} & \text{minimize } cx & \\ \text{subject to: } & x(\delta(v)) = 2 & \text{for all } v \in V, \\ & 0 \leq x_e \leq 1 & \text{for all } e \in E. \end{aligned}$$

repeat

repeat

solve current LP

contract paths which consist of edges of weight one to single 1-edges

compute minimum cut in the support graph of the current solution

if value of minimum cut < 2

add connected component cuts

add other violated cuts found using the heuristics described in Section 5.1

if the support graph corresponding to the current solution is connected

add cuts found using the separation algorithm described in Chapter 1

if no cuts added or fifth pass through loop

verify edges in reserve graph to see if any of them need to be added

else

verify edges in reserve graph to see if any of them need to be added

if cuts added

solve current LP

of the newly added constraints, remove the ones which are non-active

until no cuts added and no edges added

check remaining edges to see if they need to be added

until no edges added

Chapter 6

Computational Results

In this chapter we test the effectiveness of the decomposition algorithm and the algorithm for finding violated subtour constraints which is described in this thesis. Moreover, in Section 6.4 of this chapter, we compare our algorithm for finding violated subtour constraints with the cut generator implemented in CONCORDE. CONCORDE is a very powerful computer code designed specifically to work with the TSP. It was developed by David Applegate, Robert Bixby, Vasek Chvátal and Bill Cook and it can be freely downloaded for acedemic purposes from http://www.keck.caam.rice.edu/concorde.html.

The tests performed were executed on 50 two-dimensional Euclidean TSP instances obtained from TSPLIB (located at http://www.iwr.uniheidelberg.de/iwr/comopt/ software/TSPLIB95). TSPLIB is a repository for several different kinds of TSP which come from real world problems. The naming convention for these problems is as follows. The first part of the name holds the initials of the creator(s) of the problem and the numerical part of the name contains the number of nodes in the problems. Note that the 50 problems we looked at ranged from 101 to 18512 nodes.

The testing was done on a SunW UltraSPARC-II Server running Unix with a single 400 Mhz processor. The machine had 2 gigabytes of memory and CPLEX 7.0 was used

as the LP solver.

Note that CONVCOMB-SEP was implemented on a machine running only a single processor. However, as we have illustrated throughout this thesis, the efficiency of CONVCOMB-SEP can be improved by working in a parallel environment. Since the main goal of this research was to develop a new parallel cutting plane approach for solving the Subtour Elimination Problem, the times reported for CONVCOMB-SEP are the estimated times we would have achieved should we have worked in a parallel environment. To compute these estimated running times, we have assumed the following:

1. Given a point z in Q_F^n, we can use several processors in parallel to compute a list of violations in each vertex x present in the decomposition of z (see Chapter 1).

2. We can use several processors in parallel to find the violations of type B in any vertex x of Q_F^n (see Section 4.2.2).

3. The second heuristic presented in Section 5.1 is also implemented in a parallel environment.

Note that all times reported in this chapter are given in seconds.

6.1 Testing the effectiveness of the enhancements presented in Chapter 3

In this section we test the effectiveness of the three enhancements presented in Chapter 3 to speed up the decomposition process. These three enhancements, which we will label E1, E2 and E3 for convenience, can be summarized as follows:

Enhancement E1 (see Section 3.2.1): To find a convex combination for any point $z \in Q_F^n$, we can fix all the 0 and 1 components of z and simply find a convex combina-

tion of the vector corresponding to the fractional support graph of z.

Enhancement E2 (see Section 3.2.2): Given any point z in Q_F^n, in order to find a convex combination for z, it is sufficient to find a convex combination for each of the vectors corresponding to the connected components of the fractional support graph of z.

Enhancement E3 (see Section 3.2.3): Given any point z in Q_F^n, whenever a connected component of the fractional support graph of z consists of an odd disjoint half-cycle, then we can fix the edges in this cycle to value $\frac{1}{2}$ and simply find a convex combination for the vectors corresponding to the other connected components of the fractional support graph of z.

6.1.1 Testing Enhancement E1

In order for the enhancement E1 to speed up the decomposition process, the dimension of the vector $\hat{z}$ corresponding to the fractional support graph of z has to be significantly smaller than the dimension of the original point z. In order to test the effectiveness of this enhancement, we decided to compare the dimension of z and $\hat{z}$ for every point z obtained during the cutting plane process for each of the 50 problems examined. In the table below, we summarize our test results giving, for each of these 50 problem instances, the average and largest value for the dimension of $\hat{z}$ over all points z obtained during the cutting plane process of each problem.

Problem Name	average dimension of $\hat{z}$	largest dimension of $\hat{z}$
EIL101	10	10
PR144	19	19
CH150	32	32
KROB150	29	34
PR152	28	28
U159	29	29
RAT195	49	49

KROB200	38	46
TSP225	60	63
PR226	46	66
PR264	62	110
PR299	13	13
LIN318	71	131
RD400	28	28
FL417	15	24
PR439	66	105
PCB442	36	48
U574	35	47
P654	62	77
D657	87	122
U724	85	96
RAT783	57	66
PR1002	322	358
U1060	246	246
VM1084	119	138
PCB1173	38	38
D1291	165	191
RL1323	31	50
NRW1379	300	311
FL1400	534	567
U1432	168	214
FL1577	60	169
D1655	240	310
VM1748	166	166
U1817	272	348
RL1889	165	197
D2103	130	161
U2152	627	644
U2319	711	758
PR2392	95	95
PCB3038	783	814
FL3795	249	266
FNL4461	548	588
RL5915	305	324
RL5934	316	319
RL11849	1284	1376

USA13509	1619	1710
BRD14051	2064	2151
D15112	2195	2263
D18512	2656	2709

Table 6.1: Comparing the dimension of $\hat{z}$ with the dimension of z

Recall that the dimension of a point z in Q_r^n is $n(n-1)/2$. Hence, looking at these values, we can conclude that in practice, given a point z obtained during the cutting plane process, the edges in the fractional support graph corresponding to z represent only a small fraction of the original edge-set of z. In every example we considered, fixing the zero and one edges allowed us to reduce the dimension of the points we worked with from $O(n^2)$ to $O(n)$. More specifically, in every example considered, we found that fixing the zero and one edges allowed us to reduce the size of the vectors by over 99%. It is thus clear that fixing the zero and one edges in z does in fact help reduce the size of the vectors. Therefore, in practice, this enhancement allows us to considerably reduce the search space for the LP solver in Phase II of the decomposition algorithm (see Section 3.1).

6.1.2 Testing Enhancement E2

In this enhancement we consider the connected components of the fractional support graph of z. Note that if the fractional support graph of z has more than one component, then this enhancement allows us to reduce the size of the LPs in Phase I and II of the algorithm described in Section 3.1. In the column labeled 'avg. # of components' of Table 6.2, we give, for each of the 50 problems we looked at, the average number of connected components in the fractional support graph of z over all points z obtained during the cutting plane process for this problem. Moreover, in the column labeled 'max # of edges in component' of Table 6.2, we give the maximum number of edges

over all components of the fractional support graph of z over all points z considered. For comparison purposes, in the last column of this table, we also report the largest dimension of $\hat{z}$ over all the points obtained during the cutting plane process for this problem. Recall that $\hat{z}$ is the vector corresponding to the fractional support graph of z.

Problem Name	avg. # of components	max. # of edges in component	largest dimension of $\hat{z}$
EIL101	1	10	10
PR144	1	19	19
CH150	3	22	32
KROB150	2	23	34
PR152	2	16	28
U159	1	29	29
RAT195	4	25	49
KROB200	2	23	46
TSP225	5	27	63
PR226	4	33	66
PR264	10	65	110
PR299	1	13	13
LIN318	3	61	131
RD400	3	12	28
FL417	1	17	24
PR439	4	43	105
PCB442	4	21	48
U574	3	21	47
P654	6	22	77
D657	6	44	122
U724	7	20	96
RAT783	3	33	66
PR1002	17	72	358
U1060	9	137	246
VM1084	8	36	138
PCB1173	4	11	38
D1291	13	62	191
RL1323	4	11	50
NRW1379	18	58	311
FL1400	70	113	567

U1432	15	50	214
FL1577	6	69	169
D1655	14	150	310
VM1748	15	27	166
U1817	17	138	348
RL1889	9	41	197
D2103	14	17	161
U2152	29	141	644
U2319	13	607	758
PR2392	6	26	95
PCB3038	140	61	814
FL3795	25	30	266
FNL4461	46	48	588
RL5915	35	27	324
RL5934	37	30	319
RL11849	99	83	2376
USA13509	122	68	1710
BRD14051	132	88	2151
D15112	141	139	2263
D18512	180	128	2709

Table 6.2: Testing the usefulness of Enhancement E2

The number of connected components in the fractional support graph of a point z gives the number of times we have to execute the decomposition algorithm described in Section 3.1 in order to find a convex combination for z (see Section 3.2.2). Therefore, Column 2 of the above table gives us the average number of times the decomposition algorithm had to be executed in order to find a convex combination over all points z considered. Moreover, the data presented in Column 3 of this table gives us an indication of the size of the largest LP which had to be solved in Phase I and II of the algorithm described in Section 3.1. It is very interesting to note that, over all the examples that we looked at, we never found a connected component with more than 607 edges (see Column 3). Hence, although we worked with some problems with more than $15,000$ nodes, by making use of the enhancement presented in Section 3.2.2, we

never had to decompose a vector of dimension larger than 607. Comparing Columns 3 and 4 of this table we find that, in practice, the enhancement presented in Section 3.2.2 does effectively allow us to considerably reduce the size of the LPs with which we work. In fact, on average, we found that Enhancement E2 allowed us to reduce the dimension of the largest vector to decompose by over 62% and for problems with more than 1000 nodes, the reduction exceeded 77%.

To further test the effectiveness of this enhancement, we decided to compare the amount of time needed to find a convex combination when all three enhancements presented in Chapter 3 were implemented with the amount of time needed when only the enhancements described in Section 3.2.1 and Section 3.2.3 were considered. In the table below, we summarize our test results, giving for each of the 50 problems we looked at, the average amount of time in seconds required to compute the convex combination over all points z obtained during the cutting plane process of each problem. The column labeled 'E1 & E3' gives the average amount of time when Enhancements E1 and E3 are considered and the column labeled 'E1, E2 and E3' gives the average amount of time when all three enhancements are considered. Moreover, in Table 6.3, we also give the average number of vertices present in the convex decomposition over all points z obtained during the cutting plane process for each of the 50 problems we studied. More specifically, in the column labeled 'Avg. # of vertices for E1 & E3', we give the average number of vertices when only Enhancements E1 and E3 are considered and in the column labeled 'Avg. # of vertices for E1, E2 and E3', we give the average number of vertices when all three enhancements are considered. In Column 4 of Table 6.3 we have computed the amount of time gained or lost by making use of Enhancement E2 along with Enhancements E1 and E3 as a percentage of the time required when only Enhancements E1 and E3 are considered (a positive percentage indicates a reduction of the time required and a negative percentage indicates an augmentation of the time required). In Column 7 we computed the difference in the number of vertices between

both implementations as a percentage of the number of vertices when only Enhancements E1 and E3 are considered. Note that a positive percentage indicates a reduction in the number of vertices and a negative percentage indicates an augmentation in the number of vertices.

	Time (s)			Avg. # of vertices		
Problem Name	E1 & E3	E1, E2 & E3	%	E1 & E3	E1, E2 & E3	%
EIL101	0.00	0.00	0.00	2	2	0.00
PR144	0.01	0.01	0.00	3	3	0.00
CH150	0.01	0.01	0.00	3	2	33.33
KROB150	0.01	0.01	0.00	3	2	33.33
PR152	0.01	0.01	0.00	2	2	0.00
U159	0.01	0.01	0.00	2	2	0.00
RAT195	0.01	0.02	-100.00	5	4	20.00
KROB200	0.01	0.01	0.00	3	3	0.00
TSP225	0.02	0.02	0.00	3	2	33.33
PR226	0.02	0.01	50.00	6	6	0.00
PR264	0.02	0.02	0.00	3	4	-25.00
PR299	0.01	0.01	0.00	3	3	0.00
LIN318	0.03	0.01	66.67	6	4	33.33
RD400	0.01	0.01	0.00	4	4	0.00
FL417	0.01	0.00	100.00	2	2	0.00
PR439	0.03	0.01	66.67	6	3	50.00
PCB442	0.01	0.01	0.00	2	2	0.00
U574	0.01	0.01	0.00	3	2	33.33
P654	0.02	0.02	0.00	2	2	0.00
D657	0.04	0.02	50.00	8	4	50.00
U724	0.04	0.02	50.00	4	4	0.00
RAT783	0.02	0.02	0.00	4	4	0.00
PR1002	0.87	0.07	91.95	37	10	72.97
U1060	0.41	0.13	68.29	30	22	26.67
VM1084	0.06	0.03	50.00	11	4	63.64
PCB1173	0.02	0.02	0.00	2	2	0.00
D1291	0.28	0.05	82.14	25	20	20.00
RL1323	0.02	0.01	50.00	3	3	0.00
NRW1379	0.44	0.06	86.36	30	10	66.67
FL1400	1.56	0.14	91.03	44	10	77.27
U1432	0.16	0.04	75.00	18	7	61.11
FL1577	0.09	0.02	77.78	7	4	42.86

D1655	0.51	0.15	70.59	26	18	30.77
VM1748	0.18	0.04	77.78	20	6	70.00
U1817	0.55	0.09	83.64	30	14	53.33
RL1889	0.14	0.03	78.57	14	5	64.29
D2103	0.13	0.04	69.23	13	5	61.54
U2152	8.10	0.16	98.02	67	20	70.15
U2319	2.22	0.45	79.73	44	24	45.45
PR2392	0.12	0.02	83.33	13	6	53.85
PCB3038	3.44	0.23	93.31	54	9	83.33
FL3795	0.73	0.08	89.04	37	7	81.08
FNL4461	3.82	0.14	96.34	61	12	80.33
RL5915	1.51	0.11	92.72	43	7	83.72
RL5934	1.77	0.10	94.35	47	8	82.98
RL11849	281.18	0.39	99.86	175	22	87.43
USA13509	424.88	0.45	99.89	203	14	93.10
BRD14051	848.3	0.52	99.94	221	27	87.78
D15112	1320.30	0.68	99.95	230	32	86.09
D18512	2152.22	0.84	99.96	266	32	87.97

Table 6.3: Testing the usefulness of Enhancement E2

As the data reported in Table 6.3 suggests, although we now have to apply the decomposition process separately to each component of the support graph corresponding to $\hat{z}$, in most examples, the amount of time saved by reducing the size of the LPs is extremely significant. In fact, on average, the times decreased by 51.24%. Moreover, the larger the size of the problem with which we work, the more significant this time difference. For example, on average, for problems with over $1,000$ nodes, the time required decreased by 81.39% and for problems with over $10,000$ nodes, the time required decreased by over 99.9%. This data also suggests that by applying the decomposition process independently to each component of the fractional support graph of z, not only do we reduce the amount of time needed to find a convex combination, but we also considerably reduce the number of vertices present in the decomposition. On average, we found that making use of Enhancement E2 along with Enhancements E1 and E3

allowed us to reduce the number of vertices by 40.19%. Moreover, as the data reported in Table 6.3 suggests, the reduction in the number of vertices becomes more and more noticeable as the size of the problems increase. For example, for problems with over 1,000 nodes, the number of vertices decreased by 61.94% and for problems with over 10,000 nodes, the number of vertices decreased by 88.47%. As we have mentioned in Chapter 3, by reducing the number of vertices present in the convex combination of a point z in Q_F^n, we also reduce the number of processors needed to find the violations in z. Moreover, from the data presented in column 5 of this table, we conclude that, in practice, the number of vertices present in the decomposition of a point z is considerably smaller than the upper bound of $n(n-1)/2$ provided by Carathéodory's Theorem (see [L1982]).

As was mentioned in Chapter 3, we can further reduce the time spent to find the convex combination by making use of several processors in parallel to compute a convex combination for each connected component of the support graph corresponding to $\hat{z}$, denoted by $G_{\hat{z}}$. However, although we worked with only one processor, in every example that we studied, finding the convex combination for every component of $G_{\hat{z}}$ never took more than one second. Hence, from our experiments, we found that making use of additional processors in this phase of the decomposition process did not really help reduce the total amount of time required by the decomposition process. However, for larger problems, the time saved by making use of several processors in parallel would most likely be more significant. For example, from the data reported in Column 2 of Table 6.2, we find that, in general, the number of connected componenents in the fractional support graph of a point z increases with the size of the problem. Thus, we can expect that for larger problems the number of components in the fractional support graphs will be well over 100. Hence, for problems which take a long time to decompose, by making use of several processors in parallel to compute a convex combination for each component of the fractional support graph, we should obtain a

considerable reduction in the time required to compute a convex combination.

6.1.3 Testing Enhancement E3

To test the effectiveness of Enhancement E3, we compared the amount of time needed to find a convex combination when all three enhancements described in Chapter 3 were considered with the amount of time required when only Enhancements E1 and E2 were considered. We also compared the amount of time needed when both E1 and E3 were considered with the amount of time required when only E1 was considered. The resulting times, in seconds, are shown in Table 6.4. In Column 4 we have computed the amount of time gained or lost by considering Enhancement E3 along with Enhancement E1 as a percentage of the time required when only Enhancement E1 is considered and in Column 7, we have computed the amount of time gained or lost by considering all three enhancements as a percentage of the time required when only Enhancements E1 and E2 are considered. A negative percentage indicates an augmentation in the time required and a positive percentage indicates a reduction in the time required.

Problem Name	E1	E1 & E3	%	E1 & E2	E1, E2 and E3	%
EIL101	0.01	0.00	100.00	0.01	0.00	100.00
PR144	0.01	0.01	0.00	0.01	0.01	0.00
CH150	0.01	0.01	0.00	0.02	0.01	50.00
KROB150	0.01	0.01	0.00	0.01	0.01	0.00
PR152	0.01	0.01	0.00	0.01	0.01	0.00
U159	0.01	0.01	0.00	0.01	0.01	0.00
RAT195	0.02	0.01	50.00	0.02	0.02	0.00
KROB200	0.01	0.01	0.00	0.02	0.01	50.00
TSP225	0.02	0.02	0.00	0.02	0.02	0.00
PR226	0.03	0.02	33.33	0.02	0.01	50.00
PR264	0.01	0.02	-100.00	0.02	0.02	0.00
PR299	0.01	0.01	0.00	0.02	0.01	50.00
LIN318	0.04	0.03	25.00	0.02	0.01	50.00
RD400	0.02	0.01	50.00	0.02	0.01	50.00
FL417	0.01	0.01	0.00	0.02	0.00	100.00
PR439	0.04	0.03	25.00	0.03	0.01	66.67

PCB442	0.01	0.01	0.00	0.03	0.01	66.67
U574	0.02	0.01	50.00	0.03	0.01	66.67
P654	0.02	0.02	0.00	0.03	0.02	33.33
D657	0.06	0.04	33.33	0.04	0.02	50.00
U724	0.04	0.04	0.00	0.05	0.02	60.00
RAT783	0.03	0.02	33.33	0.03	0.02	33.33
PR1002	0.86	0.87	-1.16	0.10	0.07	30.00
U1060	0.52	0.41	21.15	0.16	0.13	18.75
VM1084	0.07	0.06	14.28	0.07	0.03	57.14
PCB1173	0.03	0.02	33.33	0.06	0.02	66.67
D1291	0.22	0.28	-27.27	0.06	0.05	16.67
RL1323	0.03	0.02	33.33	0.05	0.01	80.00
NRW1379	0.65	0.44	32.31	0.12	0.06	50.00
FL1400	1.61	1.56	3.11	0.15	0.14	6.67
U1432	0.24	0.16	33.33	0.09	0.04	55.56
FL1577	0.10	0.09	10.00	0.05	0.02	60.00
D1655	0.86	0.51	40.70	0.19	0.15	21.05
VM1748	0.19	0.18	5.26	0.11	0.04	63.64
U1817	0.58	0.55	5.17	0.12	0.09	25.00
RL1889	0.17	0.14	17.65	0.09	0.03	66.67
D2103	0.13	0.13	0.00	0.06	0.04	33.33
U2152	8.02	8.10	-1.00	0.17	0.16	5.88
U2319	2.66	2.22	16.54	1.23	0.45	63.41
PR2392	0.17	0.12	29.41	0.12	0.02	83.33
PCB3038	3.40	3.44	-1.18	0.23	0.23	0.00
FL3795	0.89	0.73	17.98	0.16	0.08	50.00
FNL4461	7.30	3.82	47.67	0.33	0.14	57.58
RL5915	2.10	1.51	28.10	0.25	0.11	56.00
RL5934	2.35	1.77	24.68	0.23	0.10	56.52
RL11849	375.34	281.18	25.09	0.74	0.39	47.30
USA13509	664.95	424.88	36.10	1.17	0.45	61.54
BRD14051	1254.25	848.3	32.37	1.18	0.52	55.93
D15112	1980.45	1320.36	33.33	1.37	0.68	50.36
D18512	3228.33	2154.22	33.27	1.72	0.84	51.16

Table 6.4: Testing the usefulness of the Enhancement E3

From the data reported in the above table, we find that on average, by considering Enhancement E3 along with Enhancement E1, we reduce the time required by

16.87% and by considering Enhancement E3 along with Enhancements E1 and E2, we reduce the time required by 43.34%. For problems with more than 1,000 nodes, the percentages are 19.41% and 46.08%, respectively. From this data we conclude that, in practice, fixing the odd disjoint half-cycles is beneficial.

6.2 Testing the usefulness of the algorithm presented in Chapter 4 to find violated cut constraints in a point z of Q_F^n

Let z be a point in Q_F^n and let x be a vertex of this polytope present in the convex decomposition of z. As explained in Chapter 4, the vertex x can have an exponential number of violated cut constraints. Since verifying each one of these constraints in the original point z can be impractical or even impossible with today's technology, we had to restrict ourselves to a reasonable number of violations. Consequently, the algorithm presented in Chapter 4 is not guaranteed to find every violation in the original point z. However, as we have demonstrated in the last section of Chapter 4, the algorithm described is guaranteed to find at least one violation in z if such a violation exists. In this section we test the effectiveness of the algorithm presented in Chapter 4 to find violated cut constraints in z. Recall that part of this algorithm consisted of shrinking the p components of the support graph of x to obtain weighted graphs K_p and K_{p+1}, and then using a fast minimum cut algorithm on these graphs. In particular, in this section, we test the usefulness of constructing the complete graphs K_p and K_{p+1} for finding violated cut constraints in z (see Chapter 4).

As described in Chapter 4, if the support graph of the vertex x has more than three components then we will construct the complete graph K_p where p is the number of components in G_x and compute the minimum cut in K_p. Moreover, for every cut violation of value 1 found in x, we will construct the complete graph K_{p+1} where, again, p is the number of components in G_x and compute a minimum st-cut in K_{p+1}.

Since every component of G_x can have as few as three nodes, the number of connected components in G_x can be as large as $\frac{n}{3}$. However, as we will show in Table 6.5, over all the vertices we worked with in each of the 50 Subtour Elimination Problems we looked at, the maximum number of components observed was 56. Moreover, on average, the number of components was only 0.82% of the number of nodes in the problem and the number of components never exceeded 5% of the number of nodes. Hence, from the data presented in Column 2 of Table 6.5, we conclude that, in practice, the time needed to construct the graphs K_p and K_{p+1} and compute the minimum cuts in these graphs is negligible. However, as mentioned in Chapter 4, we have to construct the complete graph K_{p+1} and compute a minimum st-cut in this graph for every cut of value 1 found in x. It is thus reasonable to assume that this extra step will be very expensive and that the number of additional violations found in z by performing this supplementary work will not justify the extra effort invested. To test this plausible concern, we decided to compare the running time of the cutting plane process when we did not construct the graphs K_p and K_{p+1} to find additional violations in z with the running time of CONVCOMB-SEP. As we have illustrated in Chapter 4, if we do not consider the graphs K_p and K_{p+1}, then it may be possible for some violations in z to remain hidden even after we consider every vertex in the convex combination of z. Thus, in the version of the cutting plane algorithm which does not consider the graphs K_p and K_{p+1}, if the cut generator ever fails to find a violation in z, we then simply use any fast minimum cut algorithm to compute the minimum cut in z, add the constraint corresponding to the cut found to the current LP and carry on with the cutting plane process. The column labelled T1 of Table 6.5 gives the running time of the cutting plane process when the graphs K_p and K_{p+1} are not considered and the column labelled T2 gives the running time of CONVCOMB-SEP for each of the 50 problems we examined. In Column 4 of Table 6.5 we have computed the amount of time gained or lost by considering the graphs K_p and K_{p+1} as a percentage of the

time required when this extra step is not considered. Note that a positive percentage indicated a reduction in the time required and a negative percentage indicates that it was more useful to not consider the graphs K_p and K_{p+1}.

Problem Name	maximum # of components	T1	T2	%
EIL101	2	0.11	0.11	0.00
PR144	2	0.68	0.68	0.00
CH150	3	0.30	0.30	0.00
KROB150	3	0.22	0.24	-9.09
PR152	4	0.67	0.64	4.48
U159	2	0.21	0.21	0.00
RAT195	4	0.36	0.29	19.44
KROB200	2	0.33	0.33	0.00
TSP225	4	0.39	0.35	10.26
PR226	7	1.4	1.51	-7.86
PR264	7	3.63	3.61	0.55
PR299	13	0.43	0.43	0.00
LIN318	10	0.82	0.80	2.44
RD400	1	0.55	0.55	0.00
FL417	1	2.28	2.28	0.00
PR439	4	1.17	1.13	3.42
PCB442	2	0.88	0.88	0.00
U574	3	1.33	1.26	5.26
P654	5	5.69	5.52	2.99
D657	4	1.81	1.70	6.08
U724	6	1.48	1.44	2.70
RAT783	4	1.32	1.31	0.76
PR1002	23	4.1	3.69	10.00
U1060	11	3.39	3.08	9.14
VM1084	6	2.99	2.84	5.02
PCB1173	1	2.27	2.27	0.00
D1291	17	7.79	7.96	-2.18
RL1323	2	3.68	3.68	0.00
NRW1379	15	3.27	3.22	1.53
FL1400	19	18.84	17.46	7.32
U1432	9	5.42	5.02	7.38
FL1577	6	53.14	53.45	-0.58
D1655	15	6.04	5.84	3.31
VM1748	3	4.56	4.41	3.29

U1817	18	7.88	6.79	13.83
RL1889	5	10.24	9.94	2.93
D2103	4	9.00	8.98	0.22
U2152	4	9.81	10.01	-2.04
U2319	33	13.11	10.06	23.26
PR2392	2	5.30	5.30	0.00
PCB3038	20	10.00	9.40	6.00
FL3795	17	65.30	63.71	2.43
FNL4461	4	16.26	15.54	4.43
RL5915	10	46.35	43.94	5.20
RL5934	9	48.00	46.66	2.80
RL11849	55	146.25	139.41	4.68
USA13509	37	103.21	99.59	3.51
BRD14051	47	100.14	94.11	6.02
D15112	50	134.60	120.86	10.21
D18512	56	174.29	152.59	12.45

Table 6.5: Testing the usefulness of constructing the graphs K_p and K_{p+1}

It is clear that not considering the graphs K_p and K_{p+1} reduces the running time of each step of the cutting process. However, from the data shown in Table 6.5, we conclude that, in most examples, incorporating this extra step into our cutting plane algorithm does effectively reduce the overall running time of the cutting plane process. In fact, in 33 out of the 50 examples considered, the time of the cutting plane process was reduced by considering this extra step. Moreover, on average, we found that by considering the graphs K_p and K_{p+1} we reduce the running time of the cutting plane process by 3.63%. For problems with more than 1000 nodes the improvement was of 5.00%.

6.3 Other attempts to speed up CONVCOMB-SEP

Before we test the efficiency of our cutting plane framework, CONVCOMB-SEP, we first report on some other ideas we attempted which were unsuccessful in speeding up

our algorithm for finding violated cut constraints. The tests performed in this section were executed on 12 randomly selected problems out of the 50 examples considered.

Recall from Section 5.1 that, given any point z obtained during the cutting plane process, our algorithm for finding violated cut constraints in z first executes the two heuristics described in Section 5.1. Moreover, if the support graph of z is connected, our cut generator proceeds to find additional violations by making use of the decomposition algorithm described in Chapter 1. One of the things we tried in order to speed up the cutting plane process was to only make use of the decomposition algorithm when both of the heuristics described in Section 5.1 failed to find a violated cut constraint in z. The reasoning behind this experiment is that, since the decomposition algorithm is, in general, much slower than the two heuristics described in Section 5.1, we thought that it might be worthwhile to only make use of it when necessary. In the column labeled 'T1' of Table 6.6, we report the times achieved by the cutting plane process when this new cut generator is used. For comparison purposes, in the last column of Table 6.6, we report the times achieved with our original cutting plane framework, CONVCOMB-SEP. Comparing Columns 2 and 6 of this table we find that, in general, it is more profitable to run the decomposition algorithm described in Chapter 1 together with the two heuristics described in Section 5.1 when the support graph of z is connected. In seven out of the 12 examples that we looked at, the time reported by CONVCOMB-SEP was less than the times reported in column 'T1'. Moreover, on average, we found that by only making use of the decomposition algorithm when both of the heuristics described in Section 5.1 failed to find a violation, we increased the time needed by the cutting plane process by 10.28%.

Another idea that we thought might be worthwhile was to make use of the decomposition algorithm along with the two heuristics described in Section 5.1 at every step of the cutting plane process, rather than only use the decomposition algorithm when the support graph of z was connected. We therefore modified our cut generator so

that it would reflect this modification. In the column labeled 'T2' of Table 6.6, we report the times achieved by the cutting plane process when this new cut generator is employed. Comparing the times reported in columns 3 and 6 of Table 6.6, we confirm that it is more profitable to only run the decomposition algorithm when the support graph of z is connected. In fact, in all but one examples, running the decomposition algorithm at every step of the cutting plane process increased the total time needed by the cutting plane process. On average, we found that such a modification increased the time needed by the cutting plane process by over 20%.

Moreover, to try to further reduce the time needed by the cutting plane process, we decided to modify the decomposition algorithm so that, given a point $z \in Q_F^n$, when the minimum cut in z is strictly smaller than 1, only the cuts of value 0 in the vertices present in the decomposition of z would be considered. The reasoning behind this modification is that, since any cut in a vertex of the fractional 2-factor polytope either has value 0 or a value greater than or equal to 1, if the minimum cut in z has value less than 1, then this cut must have value 0 in at least one vertex present in the decomposition of z. Hence, if the minimum cut in z has value less than 1, then by only considering the cuts of value 0 in the vertices present in the decomposition of z, we are sure to find at least one violation in z. We therefore modified our cut generator so that, when the decomposition had to be executed, if the minimum cut in z had value strictly less than 1, only the cuts of value 0 in the vertices present in the decomposition of z would be considered. Note that if the minimum cut in z had value greater than or equal to 1, then the cut generator remained unchanged. In the column labeled 'T3' of Table 6.6, we report the times achieved by the cutting plane process when this new cut generator is employed. Comparing Columns 4 of 6 of Table 6.6, we find that it is more effective to execute the decomposition algorithm as it is originally described in Chapter 4. In 10 out of the 12 examples this modification increased the time needed by the cutting plane process. Moreover, on average, we found that this modification

increased the time by 3.00%.

Recall from Section 5.1 that, at any step of the cutting plane process, if we obtain a point $z \in Q_F^n$ whose support graph is disconnected, then any set of connected components of G_z defines a violated constraint in z, giving a number of violated constraints exponential in the number of components. Since we cannot possibly consider an exponential number of constraints, in the first heuristic described in Section 5.1, we restrict ourselves to the violations corresponding to each component of G_z. By doing so, we force every component of G_z to join with another one and, consequently, we reduce the number of components by at least half. To try to reduce the number of components even more, we decided to consider a different heuristic for selecting which violated constraints to add to the current LP. Given any point $z \in \mathbb{R}^E$ whose support graph G_z is disconnected, it is clear that if we let p denote the number of components in G_z and if we let S_i represent the node-set of component i of G_z, then for each integer value of k between 1 and $p-1$, inclusively, the subtour constraint $\delta(\bigcup_{j=1}^{k} S_j)$ is violated by z. In this new heuristic procedure, we restrict ourselves to these violated constraints. Note that this heuristic gives us a different approach to considering the connected component cuts. In the column labeled 'T4' of Table 6.6, we report the times achieved by our cutting plane framework when this modification is applied. Surprisingly, adding the connected component cuts in this way seemed to considerably increase the running time of the cutting plane algorithm. In fact, on average, we found that by considering the connected component cuts in this way increased the time required by the cutting plance process by over 1000%. Moreover, note that this modification increased the total time required in each one of the 12 problems considered.

Problem Name	T1	T2	T3	T4	CONVCOMB-SEP
TSP225	0.32	0.52	0.39	0.73	0.35
LIN318	0.74	1.01	0.81	1.77	0.80
D657	1.55	2.25	1.76	4.73	1.70
U1060	3.33	4.86	3.16	60.38	3.08

FL1400	17.00	19.57	16.81	102.84	17.46
RL1889	10.94	11.96	10.25	30.62	9.94
U2319	19.22	7.01	9.92	17.7	10.06
FL3795	62.92	84.01	65.41	385.08	63.71
FNL4461	17.57	17.36	16.16	718.64	15.54
RL5915	52.17	56.32	45.57	796.15	43.94
RL11849	146.49	146.83	142.73	3341.05	139.41
D15112	128.29	127.04	128.65	2956.46	120.86

Table 6.6: Unsuccessful attemps to speed up CONVCOMB-SEP

6.4 Testing the effectiveness of our algorithm for finding violated subtour constraints

In Table 6.7 we compare the running time of CONVCOMB-SEP with the running time of CONCORDE for solving the Subtour Elimination Problem corresponding to each of the 50 examples we considered. In Column 4 of Table 6.7 we compute the time difference between the two implementations as a percentage of the running time of CONVCOMB-SEP. A positive percentage indicates that CONVCOMB-SEP was faster than CONCORDE for the particular problem and a negative percentage indicates the opposite.

Problem Name	Running time of CONVCOMB-SEP	Running time of CONCORDE	%
EIL101	0.11	0.04	-63.64
PR144	0.68	0.67	-1.47
CH150	0.30	0.16	-46.67
KROB150	0.24	0.20	-16.67
PR152	0.64	0.62	-3.13
U159	0.21	0.19	-9.52
RAT195	0.29	0.23	-20.69
KROB200	0.33	0.23	-30.30
TSP225	0.35	0.26	-25.71

PR226	1.51	0.82	-45.70
PR264	3.61	0.82	-77.29
PR299	0.43	0.37	-2.58
LIN318	0.80	0.69	-13.75
RD400	0.55	0.40	-27.27
FL417	2.28	2.09	-8.33
PR439	1.13	1.01	-10.62
PCB442	0.88	0.28	-68.18
U574	1.26	0.92	-26.98
P654	5.52	5.37	-2.72
D657	1.70	1.03	-39.41
U724	1.44	1.16	-19.44
RAT783	1.31	0.89	-32.06
PR1002	3.69	2.06	-44.17
U1060	3.08	3.71	20.45
VM1084	2.84	2.07	-27.11
PCB1173	2.27	1.32	-41.85
D1291	7.96	5.09	-36.06
RL1323	3.68	4.86	32.07
NRW1379	3.22	1.80	-44.10
FL1400	17.46	13.48	-22.79
U1432	5.02	2.55	-49.20
FL1577	53.45	42.98	-19.59
D1655	5.84	5.43	-7.02
VM1748	4.41	3.41	-22.68
U1817	6.79	5.12	-24.59
RL1889	9.94	12.14	22.13
D2103	8.98	7.11	-20.82
U2152	10.01	6.62	-33.87
U2319	10.06	1.61	-84.00
PR2392	5.30	5.36	1.13
PCB3038	9.40	5.86	-37.66
FL3795	63.71	139.61	119.13
FNL4461	15.54	10.83	-30.31
RL5915	43.94	31.69	-27.88
RL5934	46.66	39.02	-16.37
RL11849	139.41	83.04	-40.43
USA13509	99.59	101.01	1.43
BRD14051	94.11	68.18	-27.55

D15112	120.86	70.48	-41.68
D18512	152.59	84.23	-44.80

Table 6.7: Running time of CONVCOMB-SEP and CONCORDE

As the data reported in Table 6.7 suggests, CONCORDE's cutting plane algorithm is, in general, faster then CONVCOMB-SEP to solve Subtour Elimination Problems. In fact, in 44 out of the 50 problems, CONCORDE had a faster running time than CONVCOMB-SEP and, on average, we found that CONCORDE was 22.81% faster than CONVCOMB-SEP. However, it is important to realize that all aspects of the cutting plane process are being compared here, and so this comparision does not necessarily reflect the effectiveness of only our new cut generator. Since the main goal of this thesis was to develop a new parallel algorithm for finding violated cut constraints, we wanted to test the efficiency of our decomposition-based separation algorithm using a more adequate and fair comparison.

To do so, we integrated our algorithm for finding violated cut constraints into CONCORDE's cutting plane framework so that CONCORDE's implementation of all other parts of the cutting plane process was used instead of ours. Recall that, given any point z in Q_F^n, the cut generator we have implemented proceeds with the following approach to find violated cut constraints in z. The cut generator first verifies if z belongs to Q_S^n by running any fast minimum cut algorithm on the support graph corresponding to the vector z, denoted by G_z. If the value of the minimum cut in G_z is greater than or equal to 2, then z violated no subtour elimination constraints so there is no need to search for violations in z. Otherwise, the cut generator verifies the connectivity of the graph G_z. If this graph is disconnected then every connected component of G_z defines a violated subtour constraint in z. These violations are thus added to the set of violations found. Then, for every connected component of G_z, the cut generator makes use of the second heuristic described in Section 5.1 to find

additional violations in z. Every additional violation found is added to the list of violations found. Moreoever, if G_z is connected then the cut generator proceeds with the decomposition-based algorithm described in Chapter 1 to find additional violations in z.

In this section, we refer to the "modified" version of CONCORDE as the version of CONCORDE where we have replaced its algorithm which finds violated constraints with our cut generator as described above. By comparing the running time of CONCORDE's original implementation with the running time of this "modified" version of CONCORDE for solving Subtour Elimination Problems, we will obtain a fair comparison of our cut generator against the cut generator implemented in the original implementation of CONCORDE.

In the table below, we compare the running time of CONCORDE and the "modified" version of CONCORDE for solving the Subtour Elimination Problem corresponding to each of the 50 examples looked at by us. In Columns 3 and 6 of the same table, we give the number of rounds of the cutting plane process needed to obtain the optimal subtour solution for both of these implementations. Moreover, in Columns 4 and 7, we also give the total time spent looking for violations for both implementations. Note that we have estimated the times reported for the "modified" version of CONCORDE by assuming to be working in a parallel environment. In Column 8, we have computed the amount of time gained or lost by replacing CONCORDE's cut generator with ours as a percentage of the time required by CONCORDE. Note that a negative percentage indicates a loss, i.e., CONCORDE had a faster running time than the "modified" version of CONCORDE and a positive percentage indicates a gain, i.e., "modified" CONCORDE had a faster running time than CONCORDE.

	CONCORDE			'modified' CONCORDE			
Problem name	Running Time	Nbr. of Rounds	time looking for violations	Running Time	Nbr. of Rounds	time looking for violations	%
EIL101	0.04	4	0.00	0.10	3	0.05	-150.0
PR144	0.67	18	0.07	0.36	13	0.07	46.3

CH150	0.16	8	0.02	0.36	12	0.19	-125.0
KROB150	0.20	8	0.00	0.19	8	0.05	5.0
PR152	0.62	16	0.01	0.65	18	0.15	-4.8
U159	0.19	10	0.00	0.17	9	0.05	10.5
RAT195	0.23	9	0.01	0.34	12	0.15	-47.8
KROB200	0.23	10	0.01	0.37	10	0.15	-60.9
TSP225	0.26	10	0.01	0.37	11	0.21	-42.3
PR226	0.82	16	0.05	0.95	21	0.19	-15.9
PR264	0.85	15	0.04	0.98	22	0.20	-15.3
PR299	0.37	12	0.04	0.34	9	0.13	8.1
LIN318	0.69	13	0.04	0.65	13	0.17	5.8
RD400	0.40	8	0.02	0.69	12	0.32	-72.5
FL417	2.09	27	0.09	1.84	24	0.42	12.0
PR439	1.01	16	0.05	0.96	12	0.32	5.0
PCB442	0.28	8	0.04	0.27	7	0.09	3.6
U574	0.92	11	0.04	0.77	11	0.21	16.3
P654	5.37	30	0.21	3.08	26	0.59	42.6
D657	1.03	13	0.07	1.30	14	0.57	-26.2
U724	1.16	13	0.05	0.88	9	0.36	24.1
RAT783	0.89	8	0.03	0.88	10	0.26	1.1
PR1002	2.06	13	0.10	2.37	14	0.99	-15.0
U1060	3.71	22	0.23	2.53	15	0.90	31.8
VM1084	2.07	13	0.14	2.26	14	0.99	-9.2
PCB1173	1.32	10	0.09	1.21	10	0.41	8.3
D1291	5.09	17	0.25	6.78	27	3.08	-33.2
RL1323	4.86	17	0.23	2.51	13	0.66	48.4
NRW1379	1.80	13	0.17	1.83	10	0.85	-1.7
FL1400	13.48	35	0.49	7.75	23	1.37	42.5
U1432	2.55	21	0.25	2.05	12	0.72	19.6
FL1577	42.98	85	1.33	17.57	49	3.05	59.1
D1655	5.43	19	0.28	5.77	23	1.51	-6.3
VM1748	3.41	14	0.21	2.85	13	1.02	16.4
U1817	5.12	17	0.26	4.85	16	1.26	5.3
RL1889	12.14	29	0.52	6.74	22	1.69	44.5
D2103	7.11	14	0.31	4.70	13	1.01	33.9
U2152	6.62	17	0.41	7.33	17	3.02	-10.7
U2319	1.61	9	0.19	3.16	8	1.79	-96.3
PR2392	5.36	16	0.38	3.55	11	1.06	33.8
PCB3038	5.86	13	0.38	4.87	12	1.87	16.9
FL3795	139.61	89	3.74	36.74	49	9.42	73.7
FNL4461	10.83	15	0.63	8.02	12	2.64	25.9

RL5915	31.69	29	1.76	16.16	17	3.76	49.0
RL5934	39.02	29	1.78	23.23	24	5.66	40.5
RL11849	83.04	28	3.44	57.48	23	14.95	30.8
USA13509	101.01	29	4.04	61.10	21	13.84	39.5
BRD14051	68.18	23	3.16	43.20	16	12.23	36.6
D15112	70.48	23	3.53	58.63	22	20.92	16.8
D18512	84.23	20	4.39	62.61	17	16.03	25.7

Table 6.8: Comparing the cut generators

From this data we can see that, although our algorithm for finding violated cuts is much slower than the algorithm implemented in CONCORDE, in most examples, the number of iterations of the cutting plane process is reduced by making use of our algorithm for finding cut violations (see Columns 3 and 6 of Table 6.8). This explains why, in 32 out of the 50 examples we looked at, replacing CONCORDE's cut generator with ours reduced the overall running time of the cutting plane process. Moreover, in 9 of these examples, the percentage of time gained by making use of our cut generator exceeded 40%. Over all problems considered we found that, on average, the "modified" version of CONCORDE was 2.93% faster than CONCORDE. Also note that the improvements achieved with our cut generator seemed to be more significant when the size of the problem was greater than 1000. In fact, in 20 out of the 28 problems of size greater than 1000, replacing CONCORDE's cut generator with ours reduced the overall running time of the cutting plane process. Moreover, in 7 of these 28 examples, the percentage of time gained by making use of our cut generator exceeded 40%. On average, for problem with more than 1000 nodes, we achieved a 18.81% improvement when our cut generator was considered. Moreover, for problems with more than 3000 nodes, the time improvement was of 35.54%. Since the larger problems are clearly the ones of interest, the results achieved with our new cut generator prove to be very effective. Note however that, as reported in Section 6.5, as the dimension of the problem increases, the number of processors needed to achieve the

times reported by "modified" CONCORDE also increases. To further compare these two implementations of CONCORDE, we tallied the overall running time of these 50 examples for both implementations and we found that the "modified" version of CONCORDE allowed a time improvement of 39.1% over the original implementation of CONCORDE. Again, all times reported by "modified" CONCORDE assume that we are working in a parallel environment. In the next section, we report on the exact number of parallel processors required to achieve such results.

6.5 Comments on the use of parallelism

Recall that times reported for CONVCOMB-SEP are the estimated times we would have achieved should we have worked in a parallel environment. As previously mentioned at the beginning of this chapter, to compute these estimated running times, we have assumed the following:

1. Given a point z in Q_F^n, we can use several processors in parallel to compute a list of violations in each vertex x present in the decomposition of z (see Chapter 1).

2. We can use several processors in parallel to find the violations of type B in any vertex x of Q_F^n (see Section 4.2.2).

3. The second heuristic presented in Section 5.1 is also implemented in a parallel environment.

Recall from Chapter 1 and Section 4.2.2 that the previous statements presume that, given any point $z \in Q_F^n$, we have a separate processor for each half-cycle in each vertex present in the decomposition of z. In Table 6.9, for each one of the 50 problems we have looked at, we report on the exact number of processors required to achieve the estimated times reported in this chapter, i.e., for a given problem, the number of processors reported represents the maximum number of half-cycle over all

vertices present in the decomposition of z over all points z obtained during the cutting plane process for this problem. From the data presented in Table 6.9 we find that in all problems that we have considered, the number of processors needed never exceeded 152.4% of the number of nodes in the problem and, on average, the number of processors needed was 33.5% of the number of nodes in the problem. It is also important to note that, as suggested in Table 6.9, in general, the number of processors assumed, as a percentage of the dimension of the problem, increases with the size of the problem.

Problem Name	Nbr. processors assumed
EIL101	8
PR144	14
CH150	12
KROB150	22
PR152	4
U159	12
RAT195	16
KROB200	40
TSP225	24
PR226	70
PR264	94
PR299	28
LIN318	50
RD400	56
FL417	34
PR439	96
PCB442	42
U574	42
P654	28
D657	134
U724	114
RAT783	102
PR1002	498
U1060	820
VM1084	288
PCB1173	88
D1291	268
RL1323	134

NRW1379	630
FL1400	1482
U1432	382
FL1577	242
D1655	1516
VM1748	420
U1817	664
RL1889	300
D2103	84
U2152	924
U2319	870
PR2392	556
PCB3038	1564
FL3795	662
FNL4461	3080
RL5915	1330
RL5934	1040
RL11849	7454
USA13509	9936
BRD14051	18310
D15112	23028
D18512	24008

Table 6.9: Number of processors assumed

Now, since the length of the half-cycles are not all the same, in a real parallel environment, in many instances, we could make use of a single processor to handle several half-cycles of small length without affecting the total time taken by the cut generator. To test this assumption and to study the effectiveness of "modified" CONCORDE in a parallel environment constrained by the number of processors, we decided to fix the number of processors available and estimate the amount of time required by "modified" CONCORDE to solve each of the 50 TSPLIB examples we considered. More precisely, we supposed that the number of processors was limited to 1, 10, 50, 100, 500, 1000 and 5000, and we estimated the amount of time required by "modified" CONCORDE to solve each of the Subtour Elimination Problems studied in this chapter.

The results achieved are reported in the table which follows. Clearly, if the number of processors available is greater than the number of processors assumed in Table 6.9 then the amount of time required by "modified" CONCORDE remains as reported in Table 6.8. To compute the values reported in Table 6.10, we assumed the workload was distributed to the processors on a first available first serve basis. For comparison purposes, we have also included the running time of "modified" CONCORDE when the number of processors available is unrestricted. Moreover, in the last column of Table 6.10 we have included the running time of the original version of CONCORDE.

Problem name	"modified" CONCORDE - Number of processors assumed								CONCORDE
	1	10	50	100	500	1000	5000	unlimited	
EIL101	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.04
PR144	0.36	0.36	0.36	0.36	0.36	0.36	0.36	0.36	0.67
CH150	0.36	0.36	0.36	0.36	0.36	0.36	0.36	0.36	0.16
KROB150	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.20
PR152	0.65	0.65	0.65	0.65	0.65	0.65	0.65	0.65	0.62
U159	0.17	0.17	0.17	0.17	0.17	0.17	0.17	0.17	0.19
RAT195	0.34	0.34	0.34	0.34	0.34	0.34	0.34	0.34	0.23
KROB200	0.41	0.37	0.37	0.37	0.37	0.37	0.37	0.37	0.23
TSP225	0.38	0.37	0.37	0.37	0.37	0.37	0.37	0.37	0.26
PR226	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.82
PR264	0.99	0.98	0.98	0.98	0.98	0.98	0.98	0.98	0.85
PR299	0.34	0.34	0.34	0.34	0.34	0.34	0.34	0.34	0.37
LIN318	0.65	0.65	0.65	0.65	0.65	0.65	0.65	0.65	0.69
RD400	0.77	0.69	0.69	0.69	0.69	0.69	0.69	0.69	0.40
FL417	1.84	1.84	1.84	1.84	1.84	1.84	1.84	1.84	2.09
PR439	1.06	0.96	0.96	0.96	0.96	0.96	0.96	0.96	1.01
PCB442	0.27	0.27	0.27	0.27	0.27	0.27	0.27	0.27	0.28
U574	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.92
P654	3.11	3.08	3.08	3.08	3.08	3.08	3.08	3.08	5.37
D657	1.55	1.30	1.30	1.30	1.30	1.30	1.30	1.30	1.03
U724	0.94	0.88	0.88	0.88	0.88	0.88	0.88	0.88	1.16
RAT783	0.91	0.88	0.88	0.88	0.88	0.88	0.88	0.88	0.89
PR1002	5.54	2.64	2.39	2.38	2.37	2.37	2.37	2.37	2.06
U1060	3.40	2.61	2.54	2.53	2.53	2.53	2.53	2.53	3.71
VM1084	2.94	2.30	2.26	2.26	2.26	2.26	2.26	2.26	2.07
PCB1173	1.22	1.21	1.21	1.21	1.21	1.21	1.21	1.21	1.32
D1291	13.35	7.32	6.82	6.78	6.78	6.78	6.78	6.78	5.09
RL1323	2.61	2.51	2.51	2.51	2.51	2.51	2.51	2.51	4.86

NRW1379	3.81	2.02	1.86	1.83	1.83	1.83	1.83	1.83	1.80
FL1400	8.47	7.80	7.75	7.75	7.75	7.75	7.75	7.75	13.48
U1432	2.62	2.10	2.06	2.05	2.05	2.05	2.05	2.05	2.55
FL1577	17.86	17.59	17.57	17.57	17.57	17.57	17.57	17.57	42.98
D1655	7.24	5.91	5.79	5.78	5.77	5.77	5.77	5.77	5.43
VM1748	3.60	2.91	2.85	2.85	2.85	2.85	2.85	2.85	3.41
U1817	5.85	4.90	4.85	4.85	4.85	4.85	4.85	4.85	5.12
RL1889	7.32	6.77	6.74	6.74	6.74	6.74	6.74	6.74	12.14
D2103	4.88	4.71	4.70	4.70	4.70	4.70	4.70	4.70	7.11
U2152	27.41	9.05	7.53	7.37	7.33	7.33	7.33	7.33	6.62
U2319	12.17	3.95	3.24	3.18	3.16	3.16	3.16	3.16	1.61
PR2392	3.57	3.55	3.55	3.55	3.55	3.55	3.55	3.55	5.36
PCB3038	12.03	5.56	4.99	4.92	4.87	4.87	4.87	4.87	5.86
FL3795	51.03	37.97	36.88	36.75	36.74	36.74	36.74	36.74	139.61
FNL4461	14.48	8.61	8.11	8.05	8.02	8.02	8.02	8.02	10.83
RL5915	18.68	16.36	16.18	16.16	16.16	16.16	16.16	16.16	31.69
RL5934	27.04	23.58	23.28	23.25	23.23	23.23	23.23	23.23	39.02
RL11849	381.54	89.28	63.45	60.25	57.76	57.26	57.48	57.48	83.04
USA13509	241.46	78.86	64.50	62.73	61.33	61.18	61.10	61.10	101.01
BRD14051	392.86	77.85	49.92	46.43	43.71	43.39	43.20	43.20	68.18
D15112	956.37	147.7	76.09	67.13	60.03	59.22	58.67	58.63	70.48
D18512	466.89	102.75	70.53	66.50	63.28	62.89	62.63	62.61	84.23

Table 6.10: Fixing the number of processors available

In the table which follows we report on the amount of time gained or lost by replacing CONCORDE's cut generator with ours when the number of processors is restricted as a percentage of the time required by the original version of CONCORDE for each of the 50 example we considered. Note that a negative percentage indicates that CONCORDE had a faster running time than "modified" CONCORDE and a positive percentage indicates the opposite. Examining the data reported in Table 6.11, we find that although the number of processors is restricted, the results achieved by "modified" CONCORDE are still very impressive. In fact, over all problems considered, on average, with only 50 processors available, we find that "modified" CONCORDE is 1.52% faster that CONCORDE. Moreover, with 1000 processors, the improvement is 2.89%. We also noticed that the improvements achieved by our cut generator seemed to be more

significant for bigger problems. In fact, on average, for problems with more than 1000 nodes, we achieved a 16.30% improvement and an 18.75% improvement with our cut generator when we fixed the number of processors available at 50 and 1000, respectively. Moreover, for problems with more than 3000 nodes, the time improvements were of 29.76% and 35.38%, respectively. To further analyse the effectiveness of "modified" CONCORDE when the number of processors available is limited, we decided to tally the overall running time of these 50 examples for both "modified" CONCORDE and the original version of CONCORDE and we found that, with 50 processors, "modified" CONCORDE allowed a time improvement of 33.69% and, with 1000 processors, the improvement was of 38.96%. These results clearly demonstrate the effectiveness of our new cut generator. Moreover, the results shown in Tables 6.10 and 6.11 confirm that in a real parallel environment, the number of processors could certainly be reduced from that reported in Table 6.9 without affecting the times reported in Table 6.8.

Problem	% of time gained or lost - Number of processors assumed						
name	1	10	50	100	500	1000	5000
EIL101	-150.00	-150.00	-150.00	-150.00	-150.00	-150.00	-150.00
PR144	46.27	46.27	46.27	46.27	46.27	46.27	46.27
CH150	-125.00	-125.00	-125.00	-125.00	-125.00	-125.00	-125.00
KROB150	5.00	5.00	5.00	5.00	5.00	5.00	5.00
PR152	-4.84	-4.84	-4.84	-4.84	-4.84	-4.84	-4.84
U159	10.53	10.53	10.53	10.53	10.53	10.53	10.53
RAT195	-47.83	-47.83	-47.83	-47.83	-47.83	-47.83	-47.83
KROB200	-78.26	-78.26	-78.26	-78.26	-78.26	-78.26	-78.26
TSP225	-46.15	-42.31	-42.31	-42.31	-42.31	-42.31	-42.31
PR226	-15.85	-15.85	-15.85	-15.85	-15.85	-15.85	-15.85
PR264	-16.47	-15.29	-15.29	-15.29	-15.29	-15.29	-15.29
PR299	8.11	8.11	8.11	8.11	8.11	8.11	8.11
LIN318	5.80	5.80	5.80	5.80	5.80	5.80	5.80
RD400	-92.50	-72.50	-72.50	-72.50	-72.50	-72.50	-72.50
FL417	11.96	11.96	11.96	11.96	11.96	11.96	11.96
PR439	-4.95	-4.95	-4.95	-4.95	-4.95	-4.95	-4.95
PCB442	3.57	3.57	3.57	3.57	3.57	3.57	3.57
U574	16.30	16.30	16.30	16.30	16.30	16.30	16.30
P654	42.09	42.64	42.64	42.64	42.64	42.64	42.64
D657	-50.49	-26.21	-26.21	-26.21	-26.21	-26.21	-26.21

U724	18.97	24.14	24.14	24.14	24.14	24.14	24.14
RAT783	-2.25	1.12	1.12	1.12	1.12	1.12	1.12
PR1002	-168.93	-28.16	-16.02	-15.53	-15.05	-15.05	-15.05
U1060	8.36	29.65	31.54	31.81	31.81	31.81	31.81
VM1084	-42.03	-11.11	-9.18	-9.18	-9.18	-9.18	-9.18
PCB1173	7.58	8.33	8.33	8.33	8.33	8.33	8.33
D1291	-162.28	-43.81	-33.99	-33.20	-33.20	-33.20	-33.20
RL1323	46.30	48.35	48.35	48.35	48.35	48.35	48.35
NRW1379	-111.67	-12.22	-3.33	-1.67	-1.67	-1.67	-1.67
FL1400	37.17	42.14	42.51	42.51	42.51	42.51	42.51
U1432	-2.75	17.65	19.22	19.61	19.61	19.61	19.61
FL1577	58.45	59.07	59.12	59.12	59.12	59.12	59.12
D1655	-33.33	-8.83	-6.63	-6.45	-6.26	-6.26	-6.26
VM1748	-5.57	14.66	16.42	16.42	16.42	16.42	16.42
U1817	-14.26	4.30	5.27	5.27	5.27	5.27	5.27
RL1889	39.70	44.23	44.48	44.48	44.48	44.48	44.48
D2103	31.36	33.76	33.90	33.90	33.90	33.90	33.90
U2152	-314.05	-36.71	-13.75	-11.33	-10.73	-10.73	-10.73
U2319	-655.90	-145.34	-101.24	-97.52	-96.27	-96.27	-96.27
PR2392	33.40	33.77	33.77	33.77	33.77	33.77	33.77
PCB3038	-105.29	5.12	14.85	16.04	16.89	16.89	16.89
FL3795	63.45	72.80	73.58	73.68	73.68	73.68	73.68
FNL4461	-33.70	20.50	25.12	25.67	25.95	25.95	25.95
RL5915	41.05	48.37	48.94	49.01	49.01	49.01	49.01
RL5934	30.70	39.57	40.34	40.42	40.47	40.47	40.47
RL11849	-359.47	-7.51	23.59	27.44	30.44	30.68	30.78
USA13509	-139.05	21.92	36.14	37.90	39.28	39.43	39.51
BRD14051	-476.21	-14.18	26.78	31.90	35.89	36.36	36.64
D15112	-1256.94	-109.56	-7.96	4.75	14.83	15.98	16.76
D18512	-454.30	-21.99	16.26	21.05	24.87	25.34	25.64

Table 6.11: Percentage of time gained or lost

In a real parallel environment, there are certainly other aspects of the cutting plane process which could be parallelized and which we were not able to take into account here. For examples, as mentioned in [LRT2001], in a real parallel configuration, each violated cut found by our cut generator could immediately be returned to the LP module, rather than being returned within a group of cuts when the function terminates. This would allow the LP to begin adding cuts and re-solve the current

relaxation before the cut generator is finished.

To estimate the times reported by "modified" CONCORDE we assumed that each processor in the parallel computing environment had the same capacity of the processor which we used to perform our computations. Moreover, as the numbers in Table 6.9 suggest, the number of processors assumed generally increases with the size of the problem. Hence for large problems, it might be very difficult to have access to such computing power. To overcome this problem, one could make use of a grid network. In a grid network, people from around the world volunteer their computers, servers and storage to create a supercomputer capable of performing trillions of calculations per second. Grids make use of latent power that, at any one time, is not being used. It can thus result in a huge gain in power and speed and hence accelerate compute-intensive processes. Moreover, any user whose desktop PC, say, is contributing processing power to the grid will experience no negative effects since the grid runs in the background, utilizing available resources when needed by the system. If the PC user decides to run an application that requires more processing power, the work currently being processed on that machine will be dynamically reallocated to another machine in the grid with available processing power. In addition to being inexpensive, such a computing system has virtually no limitations in terms of the number of processors involved.

Chapter 7

Conclusion

In this thesis we have developed a new parallel algorithm for finding violated subtour elimination constraints within a cutting plane approach for solving the SEP. Although this decomposition-based methodology for finding violated constraints had already been introduced and successfully applied to other problems in combinatorial optimization, we were able to improve the method by developing two new enhancements which help speed up the decomposition process. As we have demonstrated in Chapter 6 when we discussed the computational results, one of these enhancements greatly reduces the running time of the decomposition algorithm. It is also important to note that, as we have demonstrated in Section 3.3, these enhancements can be applied in a straightforward manner to other problems in combinatorial optimization. In Chapter 4, we were able to exploit the well-defined structure of the vertices of the fractional 2-factor polytope to develop a linear time parallel algorithm for finding the complete set of violated cut constraints in any component of the support graph of any vertex of the fractional 2-factor polytope. However, as we have demonstrated in Chapter 4, a vertex of Q_F^n can have an exponential number of violated constraints and hence, verifying each one of these constraints in a point $z \in Q_F^n$ can be impractical or even impossible with today's technology. Nevertheless, as we have demonstrated in Chapter 4, we were able

to overcome this problem and construct an exact separation algorithm for any point in Q_F^n. In Chapter 5, we described several heuristics which help speed up our cutting plane framework, CONVCOMB-SEP. Moreover, we explained how our framework had to be adapted in order to solve large-scaled SEP. Although, our cutting plane framework did not prove to be very efficient when compared to the state-of-the-art TSP code CONCORDE, the data reported in Section 6.4 demonstrate the usefulness of the decomposition-based algorithm described in this thesis for finding violated subtour constraints. As shown in Sections 6.4 and 6.5, if we assume that "modified" CONCORDE is running in a parallel computing environment with more than 50 processors, then over all 50 examples considered for the tests performed in Chapter 6 we found that, when integrated into the very powerful software package CONCORDE, our parallel algorithm for finding violated subtour constraints proved to be over 30% more effective than the original cut generator implemented in CONCORDE. Moreover, we showed that these results can be further improved by increasing the number of processors available.

We have several ideas for topics of future research, the most important of which would be to implement our algorithm for finding violated cut constraints into a parallel computing environment. In terms of testing, the results obtained in Chapter 6 have demonstrated that this decomposition-based methodology is very useful for finding violated cut constraints. However, to further substantiate our claim, we believe that our cutting plane algorithm should be implemented in a parallel environment. By doing so, we would obtain more accurate figures. Considering the communication overhead which occurs in parallel environments, we expect that once our code is implemented in a parallel environment, the times achieve by our cut generator will be slightly more than the estimations we have provided. It would therefore be very interesting to find out exactly how much of an improvement our algorithm for finding violated cut constraints can achieve in practice in a parallel environment when compared to the cut generator implemented in CONCORDE, especially for larger problems.

It would also be interesting to apply this decomposition-based methodology to other separation problems in combinatorial optimization. For example, we could make use of this decomposition idea to develop a new parallel cutting plane approach for solving the matching problem. For the matching problem, we could decompose the points obtained during the cutting plane process into a set of vertices of the fractional matching polytope and then search for violations in these vertices in parallel by making use of an algorithm which exploits the well-defined structure of the vertices of the fractional matching polytope. However, since the support graph of the vertices of the fractional matching polytope consists of several disconnected components, it is easy to see that, for this problem as well, there will be an exponential number of violations to consider. Since we cannot possibly consider an exponential number of violations, the algorithm constructed will not be guaranteed to find the complete set of violations at each step of the cutting plane process. It would be interesting to find a problem where the point obtained during the cutting plane process can be decomposed into a set of nicely structured points guaranteed to have only a modest number of violations. As we have shown in Chapter 1, if it were possible to consider the complete set of violations in each point present in the decomposition in a reasonable amount of time, then we would be able to find the complete set of violations at each step of the cutting plane process. Through our experiments with the SEP, we found that, in general, the more violations that were found at each step of the cutting plane process, the faster we obtained the optimal subtour solution. Hence, if at each step of the cutting plane process, we could find the complete list of violations, the cutting plane algorithm developed would most certainly be very efficient.

One other idea for future work would be to try applying our ideas for finding violations to other families of necessary constraints for the STSP. For example, the comb constraints (see [GP1979A] and [GP1979B] for more details) for the STSP are a set of constraints that are very useful in cutting plane approaches for solving the STSP,

yet no polynomial-time algorithms is known for the exact separation problem of these constraints. It may be possible to find good separation methods for comb constraints for the specially structured vertices of Q^n_F, and this, combined with our decomposition method described in this thesis, could lead to improved solution times for the STSP within a cutting plane framework.

References

[ABCC1995] D. Applegate, R. Bixby, V. Chvátal and W. Cook: "Finding cuts in the TSP (A preliminary report)", *DIMACS 95-05* (1995).

[B1970] M. Balinski: "On recent developments in integer programming", in H.W. Kuhn (ed.), *Proceedings of the Princeton Symposium on Mathematical Programming*, Princeton University Press, N.J., 267-302 (1970).

[B1986] S. Boyd: "The Subtour Polytope of the Travelling Salesman Problem", Doctoral Thesis, University of Waterloo, Waterloo, Ontario (1986).

[BG1982] A. Bachem and M. Grötschel: "New aspects of polyhedral theory", *Modern Applied Mathematics-Optimization and Operations Research*, North Holland, 51-106 (1982).

[BL2002] S. Boyd and G. Labonté: "Finding the exact integrality gap for small travelling salesman problems", *Integer Programming and Combinatorial Optimization 2002*, Springer-Verlag, Berlin Heidelberg, 83-92 (2002).

[C1976] N. Christofides: "Worst case analysis of a new heuristic for the traveling salesman problem", Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh (1976).

[CG1994] B.V. Cherkassy and A.V. Goldberg: "On implementing push-relabel method for the maximum flow problem", Technical Report STAN-CS-94-1523, Departement of Computer Science, Stanford University (1994).

[CLRS2001] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein: "Introduction to Algorithms" second edition, McGraw-Hill Book Company, Massachussets (2001).

[D1948] G. Dantzig: "Programming in a linear structure", Comptroller, USA, Washington, D.C. (1948).

[D1963] G. Dantzig: "Linear programming and extensions", Princeton University Press, N.J. (1963).

[DFJ1954] G. Dantzig, D. Fulkerson and S. Johnson: "Solution of large-scale travelling salesman problems", *Operations Research* 2, 393-410 (1954).

[F1902] J. Farkas: "Theorie der einfachen ungleichungen", *Journal für die reine und angewandte Mathematik* 124, 1-27 (1902).

[FT1987] M.L. Fredman and R.E. Tarjan: "Fibonacci heaps and their uses in improved network optimization algorithms", *J. ACM* 34, 596-615 (1987).

[GH1961] R. Gomory and T. Hu: "Multi-terminal network flows", *Journal of the Society of Industrial and Applied Mathematics* 9(4), 551-570 (1961).

[GLS1981] M. Grötschel, L. Lovász, and A. Schrijver: "The ellipsoid method and its consequences in combinatorial optimization", *Combinatoria* 1, 169-197 (1981).

[GP1979A] M. Grötschel and M.W. Padberg: "On the symmetric travelling salesman problem I: inequalities", *Mathematical Programming* 16, 265-280 (1979).

[GP1979B] M. Grötschel and M.W. Padberg: "On the symmetric travelling salesman problem II: lifting theorems and facets", *Mathematical Programming* 16, 281-302 (1979).

[HT1973] J.E. Hopcroft and R.E. Tarjan: "Dividing a graph into triconnected components", *SIAM journal of Computing* 2, No. 3, 135-158 (1973).

[JP1985] D.S. Johnson and C.H. Papadimitriou: "Computational Complexity", in The Traveling Salesman Problem (E.L. Lawler et al, eds.), John Wiley & Sons, Chichester, 37-85 (1985).

[JRR1994] M. Jünger, G. Reinelt and G. Rinaldi: "The Travelling Salesman Problem", *Istituto Di Analisi Dei Sistemi Ed Informatica* (1994).

[JRR1995] M. Jünger, G. Reinelt and G. Rinaldi, *Handbook on operations research and management sciences: Networks* (M. Ball, T. Magnanti, C.L. Monma and G. Nemhauser, eds.), North-Holland, 225-330 (1995)

[K1972] R. Karp: "Reducibility among combinatorial problems", in R. Miller and J. Thatcher (eds.), *Complexity of Computer Computations*, Plenum Press, New York, 85-103 (1972).

[K1996] D.R. Karger: "Minimum cuts in near-linear time" In G. Miller, editor, *Proceedings of the 28th ACM Symposium on Theory of Computing*, ACM, ACM press, 56-63 (1996).

[LRT2001] L. Ladnyi, T.K. Ralphs, and L.E. Trotter Jr.: "Branch, cut, and price: Sequential and parallel", *Computational Combinatorial Optimization*, D. Naddef and M. Juenger, eds., Springer, Berlin, 223 (2001).

[L1982] S.R. Lay, "Convex Sets and Their Applications", Wiley-Interscience, New York (1982).

[L2000] M. Levine: "Finding the Right Cutting Planes for the TSP", *The ACM Journal of Experimental Algorithmics* 5, article 6 (2000).

[M1930] K. Menger: "Botenproblem", in K. Menger, (ed.), *Ergebuisse Eines Mathematischen Kolloquiums, Heft 2*, Leipzig, 11-12 (1930).

[PR1991] M. Padberg and G. Rinaldi: "A branch-and-cut algorithm for the resolution of large-scale symmetric travelling salesman problems", *Siam Review* 33, Issue 1, 60-100 (1991).

[PS1982] C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall Inc., N.J., (1982).

[RKPT2003] T.K. Ralphs, L. Kopman, W.R. Pulleyblank and L.E. Trotter, Jr.: "On the capacitated vehicle routing problem", *Mathematical Programming*, Series B 94, 343 (2003).

[SW1990] D.B. Shmoys, D.P. Williamson: "Analyzing the Held-Karp TSP bound: A monotonicity property with application", *Inf. Process. Lett.* 35, 281-285 (1990).

[W1980] L.A. Wolsey: "Heuristic analysis, linear programming and branch and bound", *Mathematical Programming Study* 13, 121-134 (1980).