INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

USING DP-CONSTRAINTS TO OBTAIN IMPROVED TSP SOLUTIONS

by

Danielle Vella

A thesis submitted in conformity with the requirements for
the degree of Master's of Science

School of Information Technology and Engineering

University of Ottawa
Ottawa, Canada.
May, 2001.

TABLE OF CONTENTS

ABSTRACT

The Travelling Salesman Problem (TSP) is a well-known NP-hard problem. Although it is unlikely that an efficient algorithm will ever be found for the TSP, some success has been achieved for large real-world instances by using the branch and cut technique. This technique requires knowledge of classes of useful valid inequalities for the polytope associated with the TSP, as well as efficient separation routines for these classes of inequalities.

The DP-constraints are a recently discovered class of valid inequalities for the TSP which contain the famous comb inequalities. An efficient separation routine is known for these constraints if certain conditions are satisfied by the point to be separated. This separation routine has never been implemented or tested. We present several performance enhancements that we made to the existing separation routine for these constraints and discuss our implementation of this improved algorithm. Our implementation runs in $O(n^3 \lg n)$, where n is the number of vertices in the graph. We also show how the outcome of the separation routine can be translated into a DP-constraint that is in a suitable form for testing. We test our implementation and provide results that we believe show the usefulness of these inequalities and the separation routine within a branch and cut framework for the TSP.

ACKNOWLEDGEMENTS

I would like to express my sincerest thanks to my supervisor, Dr. Sylvia Boyd, who has taught me and contributed so much towards the understanding of this work. Her patience and perseverance in tackling some very difficult problems encountered along the way are greatly appreciated.

A special word of thanks goes to Chris Lankester for his assistance in dealing with some weird "quirks" in the C programming language.

Chapter 1

Introduction

A salesman wants to travel to a set of cities, visiting each exactly once, and return back to his starting point in the cheapest possible way. This problem, known as the *Travelling Salesman Problem* (*TSP*), is among the most widely studied topics in computer science, with its origins dating back to the 1920s [ABCC95]. Applications of the TSP include printed circuit board production, vehicle routing and job sequencing. Despite the efforts of many, no polynomial-time algorithm is known for solving the TSP. In fact, it belongs to the class of *NP-hard* problems, and thus it is considered highly unlikely that a polynomial-time algorithm will ever be found for this problem.

The TSP can be modelled graphically as follows. Let $K_n = (V, E)$ be the complete graph with vertex set V representing the set of n cities and edge set E. Associated with each edge $e \in E$ is a cost of travel, c_e. The TSP is to find a *tour*, that is a cycle that passes through every vertex (or node) exactly once, of minimum cost. Several variations of the TSP exist; [MO] gives good descriptions and useful references for the different kinds of TSPs. For the TSP, if n is the number of vertices, there are (n-1)! / 2 possible tours. Thus, exhaustively searching for the minimum cost tour is far too time-consuming for large n. Consequently, more practical methods have evolved and been employed through the years.

Combinatorial optimization is a developing field of study that combines techniques from applied mathematics and operations research. An important advance in TSP research came in 1954 when Dantzig, Fulkerson and Johnson [DFJ54] applied ideas from combinatorial optimization to the TSP. They formulated the TSP as a 0-1 integer linear programming (ILP) problem by associating with each $e \in E$, a binary edge variable x_e such that $x_e = 1$ if e is in the tour and 0 if e is not part of the tour. Given any $S \subset V$, $\delta(S)$ refers to the set of edges with exactly one end in S. For any $L \subseteq E$, x(L) is equal to $\sum (x_e: e \in L)$. The ILP is:

$$\text{Minimize } \sum (c_e x_e: e \in E) \tag{1.1}$$

Subject to

$$x(\delta(v)) = 2, \forall v \in V \tag{1.2}$$

$$x(\delta(S)) \geq 2, \forall S \subset V, \varnothing \neq S \neq V \tag{1.3}$$

$$0 \leq x_e \leq 1, \forall e \in E \tag{1.4}$$

$$x_e \text{ integer}, \forall e \in E \tag{1.5}$$

The objective function (1.1) is to minimize the sum of the edge variables with respect to the edge costs. Equations (1.2) are the *degree constraints*. They state that the degree at each vertex is 2. Inequalities (1.3) are known as the *subtour elimination constraints*, or just *subtour constraints*. They eliminate the union of disjoint cycles as a possible solution for the above ILP. The subtour constraints say that for any set S, $1 \leq |S| \leq |V| - 1$, a tour must enter and leave S at least once, and therefore will contain at least two edges from $\delta(S)$. Inequalities (1.4) are the *upper* and *lower bounds*, while (1.5) are the *integrality constraints*.

Together, constraints (1.4) and (1.5) state that each $e \in E$ is either in the tour or not in the tour.

Given a set S of vectors, the *convex hull of 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 exist non-negative scalars $\lambda_1, \lambda_2, \ldots, \lambda_k$ and vectors $x_1, x_2, \ldots, x_k$ in S such that $\sum_{i=1}^{k} \lambda_i = 1$ and $w = \sum_{i=1}^{k} \lambda_i x_i$. By taking the convex hull of all $x \in \Re^E$ which satisfy (1.2) - (1.5), we obtain the *travelling salesman polytope*, denoted by *TSP(n)* for $n = |V|$. Thus, we can also express the TSP as (minimize cx: $x \in$ TSP(n)). Note that as with any polytope, it should be possible to give a linear programming (LP) formulation for TSP(n), i.e. describe it as the solution set of a finite system of linear inequalities and equations. However, such a complete system is not known for general n for TSP(n), although some of the necessary inequalities are known.

An important relaxation of the TSP is obtained by relaxing the integrality restriction (1.5) and only optimizing over (1.2) - (1.4). This LP problem is known as the *subtour elimination problem*, or simply *subtour problem*, and is denoted by *SEP*. The corresponding polytope, which is the convex hull of all solutions to (1.2) - (1.4), is called the *subtour elimination polytope*, and is denoted by *SEP(n)*, where $n = |V|$. Note that because there is an exponential number of subtour constraints (1.3), SEP cannot be solved practically using general linear programming techniques such as the simplex method.

Since SEP is a relaxation of TSP, we have TSP(n) $\subseteq$ SEP(n), and (min cx: x $\in$ SEP(n)) $\leq$ (min cx: x $\in$ TSP(n)). Thus, solving the SEP provides a lower bound on the optimal value for the TSP. Sometimes when optimizing over SEP(n) we might obtain a solution that is integral, in which case this solution represents an optimal tour for the TSP. In fact, for n $\leq$ 5, this will always be the case [JRR95]. In general, however, we may not be this lucky. Consider, for example, the TSP problem on 6 vertices shown in Figure 1.1(a), where the edge costs c_e for edges shown are as indicated, and the edge costs for the edges not shown are some very large number M. For this example, an optimal solution x* for the SEP is as shown in Figure 1.1(b), and has weight 9.

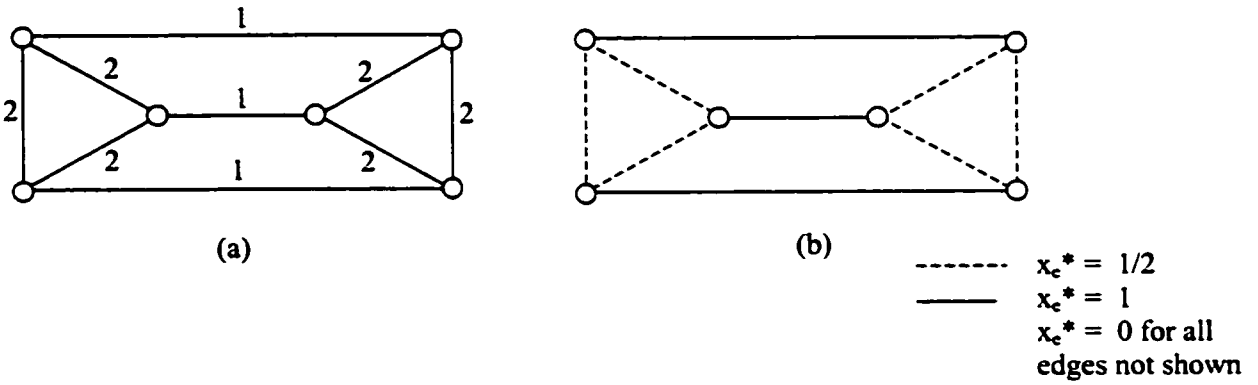


Figure 1.1: A TSP problem on 6 vertices

It is clear from the example in Figure 1.1 that for n $\geq$ 6, the linear constraints (1.2) - (1.4) are not sufficient to describe TSP(n), and thus other linear constraints must be added to obtain an LP-formulation for TSP(n). Any such constraints that are not only valid for TSP(n), but are also necessary in any linear description of it, are called *facet-inducing*.

There has been a great deal of research over the past 25 years which has focused on finding classes of facet-inducing inequalities for TSP(n) (see [JRR95] for a summary of this research). One of the first and best-known families of such facet-inducing inequalities is the class of comb inequalities of Grötschel and Padberg ([GP1-79] and [GP2-79]). A *comb* consists of a *handle* $H \subset V$, and an odd number of pairwise-disjoint *teeth* $T_j \subset V$, $j = 1, 2, \ldots, p$, $p \geq 3$. In addition, the handle must have a non-empty intersection with each tooth, but not contain any teeth. Given a comb, the corresponding *comb inequality* is

$$x(\delta(H)) + \sum_{j=1}^{p} x(\delta(T_j)) \geq 3p + 1 . \qquad (1.6)$$

Note that in the case where all the teeth of the comb consist of exactly two vertices, the corresponding constraint (1.6) is called a *2-matching inequality*, discovered by Edmonds [E65].

If we return to Figure 1.1, it turns out that the optimal solution x^* for SEP(6), shown in Figure 1.1(b), can be "cut off" by the comb inequality with handle H and teeth T_1, T_2 and T_3, depicted in Figure 1.2. In this example, the left-hand side of (1.6) is 9, while the right-hand side is 10. Thus, the point x^* violates this comb inequality by 1. This illustrates how, for $n \geq 6$, adding the comb inequalities to the inequalities (1.2) - (1.4) cuts off portions of SEP(n), resulting in a new polytope Q(n) such that $TSP(n) \subset Q(n) \subset SEP(n)$. In fact, for $n = 6$, it was shown in [BC91] that $Q(n) = TSP(n)$.

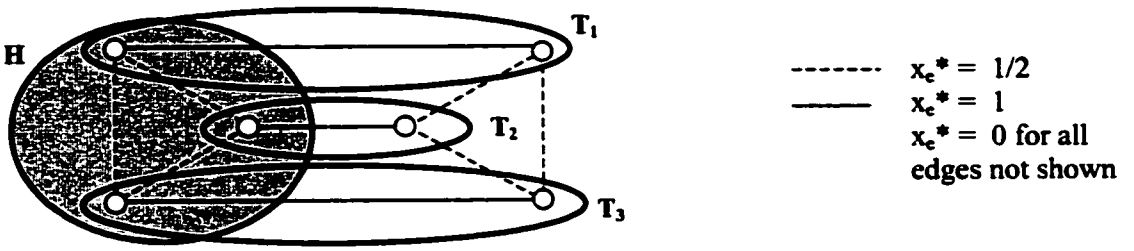


Figure 1.2: A violated comb inequality

The ideas just discussed are the basis of the very successful *cutting plane* and *branch and cut* algorithms for the TSP. The general framework of such algorithms for the TSP is as follows. First, use standard linear programming techniques to optimize (1.1) over the constraints (1.2) and (1.4). If the optimal solution x* corresponds to a tour, stop; it is an optimal solution for the TSP. Otherwise, look for valid inequalities (such as subtour elimination constraints (1.3) or comb constraints (1.6), for example) that are violated by x*. Such inequalities are often referred to as *cuts* or *cutting planes* since they "cut off" the current solution x* (note that usually one looks for violated inequalities that are also facet-inducing, as these tend to provide good cuts). If any such violated inequalities are found, they are added to the current LP and it is resolved. This process is continued until either an optimal solution x* which represents a tour is reached, or else no violated inequalities can be found for the current x*. In the latter case, we can either branch as in standard branch and bound algorithms, or else use the final x* to hopefully provide a good lower bound for the TSP instance we are trying to solve.

The cutting plane algorithm was first applied to the TSP by Dantzig, Fulkerson and Johnson [DFJ54] in 1954, where it was used to solve a 49-city problem to optimality. More recently,

it has been used to solve much larger practical instances of the TSP to optimality (see [PR91], [JRR95] and [ABCC95], for example).

A very important part of the branch and cut method is the ability to efficiently find violated inequalities for the current solution x^*. An *exact separation algorithm* for a class of valid inequalities for TSP(n) is an algorithm that given any point x^*, either finds one or more inequalities from the class that are violated by x^*, or determines that no inequality in the class is violated by x^*. For TSP(n), polynomial-time exact separation algorithms exist for the subtour elimination constraints [PG85] and the 2-matching constraints [PR82]. Recently, much time and effort has been invested in finding a polynomial-time exact separation algorithm for the comb inequalities (1.6). In [C97], Carr provides such an algorithm for combs with a fixed number of teeth. In [ABCC95], *heuristics* (algorithms that do not guarantee optimal solutions, but strive to produce fairly good results) for comb inequality separation are discussed. Note that for points $x^* \in SEP(n)$, a comb constraint can be violated by at most 1. In [FT99], Fleischer and Tardos give a polynomial-time exact separation algorithm for the class of such maximally violated comb constraints for points $x^* \in SEP(n)$ that have a planar support graph (the *support graph of* x^* is the subgraph of $K_n=(V, E)$ induced by the edge set $E^* = \{e \in E: x_e^* > 0\}$). In [CFL00], Caprara, Fischetti and Letchford give a polynomial-time separation algorithm for the class of maximally violated *mod-2 cuts*; these are a class of valid inequalities for the TSP(n) that contain the comb inequalities.

More recently, Letchford [LE00] introduced a new class of inequalities called the domino-parity constraints (or just DP-constraints). These constraints are a superclass of the comb inequalities and a subclass of the mod-2 cuts. In this same paper, he describes a polynomial-time exact separation algorithm for the case where $x^* \in SEP(n)$ and x^* has a planar support graph. His separation algorithm has the advantage over the algorithms in [FT99] and [CFL00] in that he does not restrict the separation to only maximally violated inequalities from the class. However, it is at a disadvantage over the separation algorithm from [CFL00] for mod-2 cuts in that Letchford's algorithm requires that the current solution x^* being separated must have a planar support graph, and this will not always be the case.

There has been no previous testing of any of these three algorithms, and in particular, of the Letchford separation algorithm for domino-parity constraints. Because many inequalities in this class of constraints are not facet-inducing, some people question their usefulness as cutting planes. Furthermore, some wonder how practical Letchford's algorithm is, given that many of the solutions x^* encountered during branch and cut have non-planar support graphs.

The main objective of this thesis is to investigate the practical usefulness of the domino-parity constraints and the Letchford separation algorithm within a branch and cut framework for the TSP. Essentially, we want our research to provide information and results that would help someone decide whether or not it would be worthwhile to include these constraints as cutting planes for the TSP. To this end, we implemented Letchford's algorithm, made some performance enhancements, and used it within a software package called CONCORDE (developed by David Applegate, Robert Bixby, Vasek Chvátal and Bill Cook, it is a

powerful computer code designed specifically to work with the TSP; it can be freely downloaded for academic purposes from http://www.keck.caam.rice.edu/concorde.html).

The contents of the rest of this thesis are as follows. In Chapter 2, we describe the domino-parity constraints, as well as the polynomial-time exact separation algorithm for them, as first presented by Letchford in [LE00]. In Chapter 3, we describe the details of how we translate the final cycles found by the Letchford algorithm into domino-parity constraints that are in the correct form for use by CONCORDE. Note that in this same chapter we also describe a new method for finding a "nice" equivalent form of a domino-parity constraint, in the sense that its compatibility with CONCORDE is greatly increased. In Chapter 4, we give further details of our implementation of the Letchford separation algorithm. Here, we discuss the enhancements we made to the algorithm, the most significant of which is one that often allows us to make use of Letchford's algorithm in situations where the current solution x^* does not have a planar support graph. In these cases, our version of the algorithm continues where normally the original version would halt. In Chapter 5, we report our test results, showing that the enhanced separation routine performed well when compared to the results obtained by CONCORDE. Finally in Chapter 6, we conclude with some closing remarks and ideas for future research.

The remainder of this chapter is devoted to notation. Note that some of these definitions have been previously stated, but we include them here for the convenience of the reader.

For any finite set E, let $\Re^E$ denote the set of all real vectors indexed by E, and let $|E|$ represent the *cardinality* of E. For $x \in \Re^E$ and $L \subseteq E$, let $x(L) = \sum (x_e: e \in L)$.

A *graph* G = (V, E) is an ordered pair, where V is a finite set of elements called *vertices*, and E is a finite set of elements called *edges* such that every $e \in E$ corresponds to two vertices in V called the *ends* of e. An edge with two ends that are the same vertex is called a *loop*. An edge with ends *u* and *v* is denoted by uv. Two vertices $u,v \in V$ are said to be *adjacent* if $uv \in V$. For any $v \in V$, the *degree of v* is $|\delta(v)|$.

Given a graph G = (V, E), for any $A \subseteq V$, let E(A) denote the set of edges in E with both ends in A and let $\delta(A)$ represent the set of edges in E with exactly one end in A. A *path of length m* in G, where m is a positive integer, is a sequence of edges $v_1v_2, v_2v_3, \ldots, v_{m-1}v_m$ in E, where $v_i \in V$, $i = 1, 2, \ldots, m$. A path is called a *cycle* if it begins and ends at the same vertex (i.e. $v_1 = v_m$). The graph G is *connected* if every pair of vertices is joined by a path. A *component* of G is a maximal connected subgraph of G. An edge $e \in E$ is called a *cut edge* if its removal from G disconnects the graph. If the edges in G are given a direction, then we call G a *digraph* or *directed graph*, and we call the edges *arcs*. An arc which originates at vertex i and ends at vertex j is denoted by the ordered pair (i, j), where i is called the *tail* of the arc and j is called the *head* of the arc. A path in a digraph is called a *dipath*. For any graph G = (V, E) and vector $x^* \in \Re^E$, the *support graph of* x^*, denoted by $G^* = (V^*, E^*)$, is the subgraph of G, with vertex set $V^* = V$ and edge set $E^* = \{e \in E \mid x_e^* > 0\}$.

A graph is called *complete* if every pair of vertices is joined by an edge. The complete graph on n vertices is denoted by K_n. For the remainder of this thesis, we use E and V to denote the edge set and vertex set of the complete graph K_n, unless otherwise stated.

Chapter 2

Letchford's Work

In [LE00], Adam Letchford presents a polynomial-time exact separation algorithm for the domino-parity constraints (or DP-constraints) that works provided that the current solution $x^* \in SEP(n)$ and has a planar support graph. A brief outline of his work follows. Note that in [LE00], a "$\leq$" form of the DP-constraints is used for part of the work, whereas we have chosen to describe all of the results from [LE00] using an equivalent "$\geq$" form of the inequality, in order to be consistent and for the ease of the reader. This "$\geq$" form can also be found in [LE00].

We begin by defining the DP-constraints, but first we require some related definitions.

Definition: A *domino*, denoted by $D = \{A, B\}$, is a set of vertices $D = A \cup B$ such that $A, B \neq \varnothing$, $D \subset V$ and $A \cap B = \varnothing$. The edge subset E(A:B) is called the *semicut* of the domino and represents the edges in K_n with one end in A and the other in B.

Definition: Given edge subsets $E_1, \ldots, E_r$, $E_i \subset E$ for $i = 1, 2, \ldots, r$, the *multiplicity* μ_e of an edge $e \in E$ is defined as the number of subsets $E_1, \ldots, E_r$ in which e appears. The subsets $E_1, \ldots, E_r$ are said to *support the cut* $\delta(H)$, $H \subset V$, if $\delta(H) = \{e \in E: \mu_e \text{ is odd}\}$.

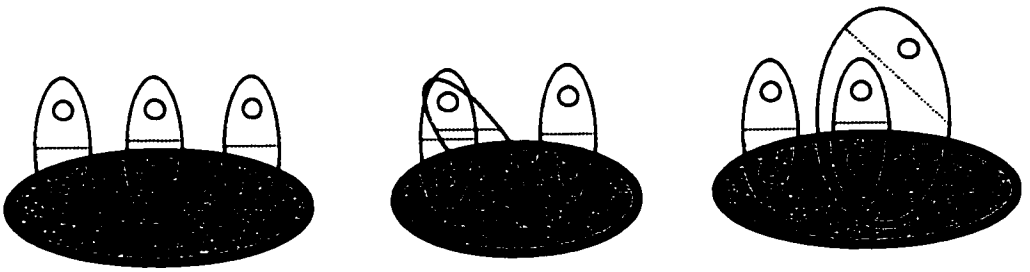
Definition: A *domino-comb* $\{F; D_1, D_2, \ldots, D_p\}$ consists of a unique edge subset $F \subseteq E$ and an odd number of dominoes $D_j = \{A_j, B_j\}$, $j = 1, 2, .., p$ such that the edge subsets F, $E(A_1 : B_1)$, $E(A_2 : B_2)$, ..., $E(A_p : B_p)$ support the cut $\delta(H)$ for some $H \subset V$. The set H is known as a *handle* of the domino-comb.

The domino-comb $\{F; D_1, D_2, \ldots, D_p\}$ can be expressed equivalently in terms of the handle H as $\{H; D_1, D_2, \ldots, D_p\}$ since given H, we can find F, and vice versa. In this latter form, the domino-comb is easier to visualize. If we divide the set F into two subsets such that $F = F_1 \cup F_2$, the relationship between F and H is:

$F_1 = \{e \in E \mid e \in \delta(H)$ and e is in an even number of the semicuts $E(A_j : B_j), j = 1, 2, \ldots, p\}$;

$F_2 = \{e \in E \mid e \notin \delta(H)$ and e is in an odd number of the semicuts $E(A_j : B_j), j = 1, 2, \ldots, p\}$.

Note that if the dominoes in a domino-comb are disjoint, $F = \delta(H) \setminus \{\bigcup_{j=1}^{p} E(A_j:B_j)\}$ and $p \geq 3$, then we actually have a comb, with each edge $e \in \delta(H)$ occurring in exactly one of the subsets F, $E(A_1 : B_1)$, $E(A_2 : B_2)$, ..., $E(A_p : B_p)$. In general, however, this is not the case for all domino-combs. In Figure 2.1, three examples of domino-combs, each having three dominoes, are shown. Unless otherwise noted, in this and subsequent figures, the handle H is shaded and the individual domino vertex sets A and B are separated by a dashed line. Note that (a) is a comb, but (b) and (c) are not.



(a) (b) (c)

Figure 2.1: Examples of domino-combs

Definition: The *DP-constraint* corresponding to a domino-comb $\{F; D_1, D_2, \ldots, D_p\}$ is

$$\sum_{j=1}^{p} x(\delta(D_j)) + \sum_{j=1}^{p} x(E(A_j{:}B_j)) + x(F) \geq 3p + 1 \,. \qquad (2.1)$$

In addition, if the DP-constraint is a comb, then this gives the (1.6) form of the comb constraint

$$\sum_{j=1}^{p} x(\delta(D_j)) + x(\delta(H)) \geq 3p + 1 \,.$$

Next, before describing Letchford's exact separation algorithm for DP-constraints, we require a few definitions pertaining to the topic of planarity.

Definition: A graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is *planar* if it can be drawn in the plane without any edges crossing; such a representation is called a *planar embedding*. If $\mathbf{G}$ is planar, then its planar embedding divides the plane into regions called *faces*.

Definition: Given a planar embedding of a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, its *planar dual* $\underline{\mathbf{G}} = (\underline{\mathbf{F}}, \underline{\mathbf{R}})$ is formed as follows. For each face of $\mathbf{G}$, create a vertex in $\underline{\mathbf{F}}$. Build the edge set $\underline{\mathbf{R}}$ for $\underline{\mathbf{G}}$ by forming an edge between two vertices u and v in $\underline{\mathbf{R}}$ whenever u and v correspond to adjacent

faces in $\mathcal{G}$ (i.e. whenever the faces corresponding to u and v share a common border). Note that the edges of $\mathcal{G}$ and $\underline{\mathcal{G}}$ are in one-to-one correspondence, and the faces of $\mathcal{G}$ and the vertices of $\underline{\mathcal{G}}$ are also in one-to-one correspondence.

Given a vector $x^* \subset SEP(n)$, let $G^* = (V^*, E^*)$ be its planar support graph and note that $V^* = V$ and $E^* \subset E$. In [LE00], Letchford describes an exact separation algorithm that finds the DP-constraint most violated by x^* (i.e. the violated DP-constraint for which the right-hand side of (2.1) minus cx^* is as large as possible), or concludes that x^* satisfies all DP-constraints; its complexity is $O(n^3)$. We explain the steps of the original algorithm and illustrate them on a small example.

Step 1: Find the red edges

Let $\underline{G}^* = (\underline{F}^*, \underline{R}^*)$ be the planar dual of $G^* = (V^*, E^*)$. We call each dual edge $r \in \underline{R}^*$ a *red edge* and assign it a weight equal to the value x_e^* of its corresponding edge e in G^*. We use our example from Figure 1.1(b) and show the red edges in Figure 2.2.

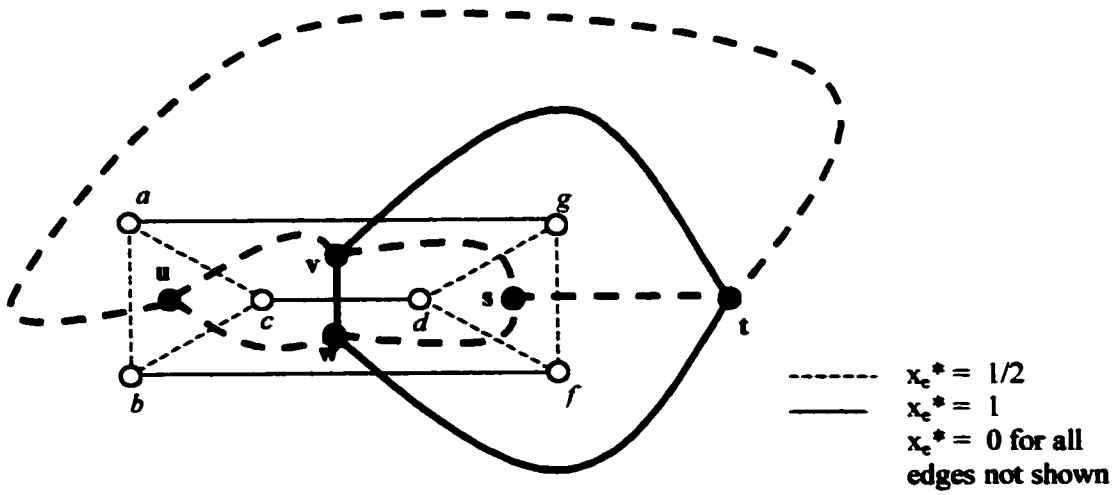


Figure 2.2: G^* and $\underline{G}^*$

Step 2: Find the green edges

For every vertex pair {s, t} in $\underline{G}^*$, we use the red edges (i.e. the edges of the dual) to find three edge-disjoint s to t paths in $\underline{G}^*$ of minimum total weight, w_{st}. If $w_{st} - 3 < 1$, we form a *green edge* of weight $w_{st} - 3$ between s and t. From the vertex set $\underline{F}^*$ and the edge set which is the union of the red and the green edges, we create a new graph $\underline{M}^*$, shown in Figure 2.3 for our example.

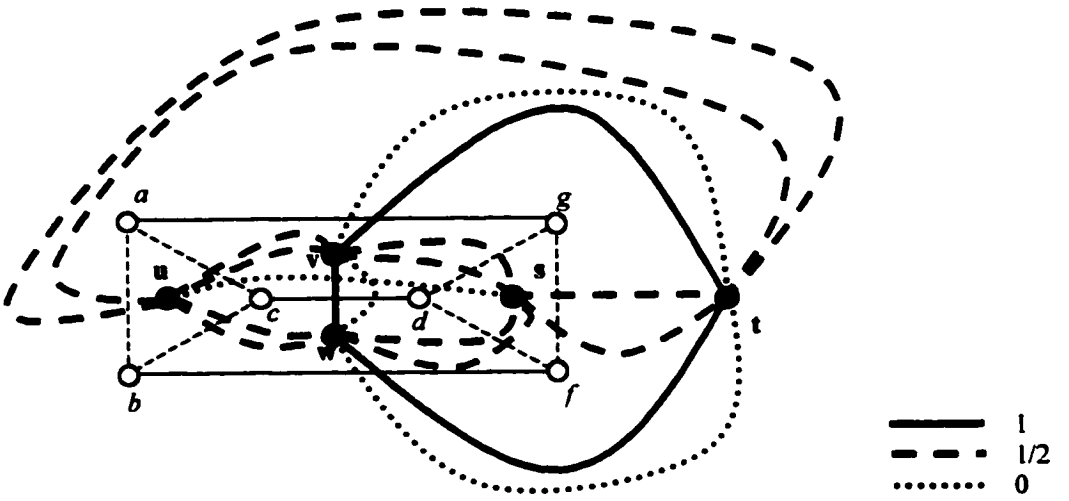


Figure 2.3: $\underline{M}^*$

Step 3: Find the minimum weight cycle in $\underline{M}^*$

Find a minimum weight cycle in $\underline{M}^*$ of weight less than 1 that uses an odd number of green edges and any number (including 0) of red edges, or determine that no such cycle exists. For our ongoing example, the reader can verify from Figure 2.3 that v, t, w, v, consisting only of green edges, forms such a cycle; it has weight 0.

It turns out that any minimum weight cycle in $\underline{M}^*$ of weight less than 1 using an odd number of green edges and any number of red edges corresponds to a DP-constraint violated by x^*. The complete proof for this can be found in [LE00], but the following brief discussion

touches on the theory behind Letchford's separation algorithm. The reader is referred to [LE00] for the proofs of any results, and for further details.

Definition: Given an inequality $ax \geq a_0$ and a vector x^* that satisfies this inequality, the *surplus* of $ax \geq a_0$ w.r.t. x^* is $ax^* - a_0$.

Definition: If $x^* \in SEP(n)$, the *weight of a domino* $D = \{A, B\}$ w.r.t. x^* is denoted by *w(A,B)*. It is the sum of one half times the surpluses w.r.t. x^* of the subtour constraints for A, B and $A \cup B$, i.e. $w(A, B) = 1/2(x^*(\delta(A)) + x^*(\delta(B)) + x^*(\delta(A \cup B))) - 3$. Note that this can be rewritten as

$$w(A, B) = x^*(\delta(D)) + x^*(E(A:B)) - 3 . \tag{2.2}$$

Using (2.2), it follows that if $x^* \in SEP(n)$, then the surplus of a DP-constraint (2.1) is

$$\sum_{j=1}^{p} w(A_j, B_j) + x(F) - 1. \tag{2.3}$$

Clearly, a DP-constraint is violated by $x^* \in SEP(n)$ if and only if its surplus (2.3) is negative, which will occur if and only if we have

$$\sum_{j=1}^{p} w(A_j, B_j) + x(F) < 1. \tag{2.4}$$

Thus all dominoes in a DP-constraint which violates some $x^* \in SEP(n)$ must have weight less than 1. Such dominoes are called *usable*.

It is shown in [LE00] that the green edges in $\underline{M}^*$ are in one-to-one correspondence with the usable dominoes for x^*. In fact, the following lemma is proved.

Lemma 2.1. If G^* is planar, $x^* \in SEP(n)$ and $D = \{A, B\}$ is a usable domino in G^*, then there are two distinct vertices s and t in $\underline{G}^*$ such that $E^*(A:B)$, $E^*(A:V\setminus(A\cup B))$ and $E^*(B:V\setminus(A\cup B))$ correspond to three edge-disjoint s to t paths in $\underline{G}^*$ of minimum total weight. Furthermore, w(A, B) can be calculated by summing the weights of the red edges in these three s to t paths and subtracting 3 from the total.

It can be seen from Lemma 2.1 that the green edges in $\underline{M}^*$ are formed exactly as described in Lemma 2.1, and thus correspond to usable dominoes. Also, the weight on each of the green edges corresponds to the domino's weight.

In order to make use of $\underline{M}^*$ to find a violated DP-constraint, the following definition and lemma from [LE00] are required.

Definition: For $E_j \subset E^*$, j = 1, 2, ..., r, the edge subsets $E_1, \ldots E_r$ *support a cut in* G^* if the edges $e \in E$ with μ_e odd form a cut in G^*.

Lemma 2.2. If $x^* \in SEP(n)$, then a DP-constraint is violated by x^* if and only if there are p usable dominoes $\{A_1, B_1\}, \ldots, \{A_p, B_p\}$ and a set of edges $R^* \subset E^*$ such that:

1) p is a positive odd integer;

2) the edge subsets $E^*(A_1:B_1), \ldots, E^*(A_p:B_p)$, R^* support a cut in G^*;

3) $\sum_{j=1}^{p} w(A_j, B_j) + x^*(R^*) < 1$.

It turns out that sets of dominoes and edges R* from Lemma 2.2 correspond to certain cycles in $\underline{M}^*$, as stated in the following lemma from [LE00].

Lemma 2.3. A DP-constraint is violated if and only if there exists a cycle in $\underline{M}^*$ of weight less than 1 such that this cycle is formed from an odd number of green edges and any number (including 0) of red edges.

Note that from Lemma 2.3 it follows that every such cycle in $\underline{M}^*$ of weight less than 1 corresponds to a violated DP-constraint. The p dominoes of the violated constraint correspond to the p green edges in the cycle, and the red edges in the cycle correspond to the edges e in F in K_n for which $x_e^* > 0$.

To illustrate these ideas, we go back to our example. Recall that in Figure 2.3 we have a cycle **v, t, w, v** of weight 0 which contains three green edges. This cycle corresponds to a DP-constraint, which x* violates by 1, with three dominoes. In Figure 2.4, we show the three dominoes $D_1 = \{a, g\}$, $D_2 = \{b, f\}$ and $D_3 = \{c, d\}$ corresponding to the green edges in the cycle **v, t, w, v** (represented by the thick lines). In Figure 2.4, we use purple for D_1, blue for D_2 and brown for D_3. Consider D_1. It corresponds to the three shortest edge-disjoint paths in $\underline{G}^*$ of minimum total weight, namely **v, s, t**, and **v, t**, and **v, u, t**. Note that the middle path **v, t** separates $A_1 = \{a\}$ from $B_1 = \{g\}$ and that dominoes D_2 and D_3 also correspond to the three shortest paths used to form the other two green edges in the cycle in $\underline{M}^*$.

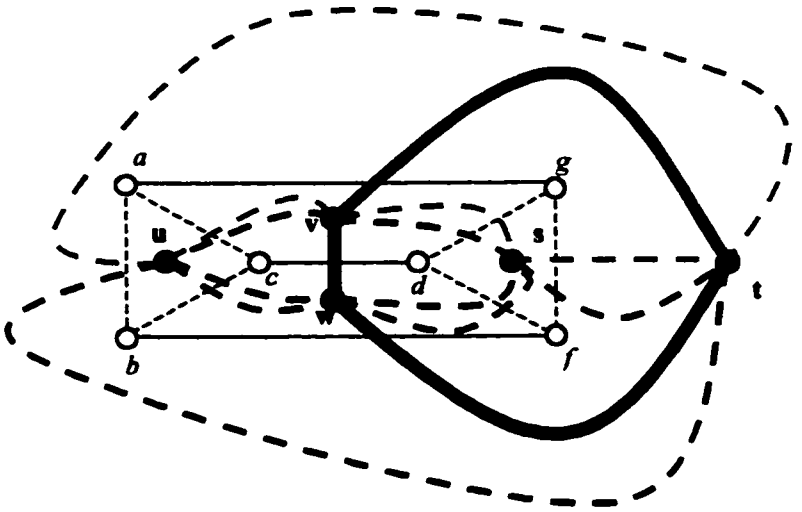


Figure 2.4: Three usable dominoes

The handle for the violated DP-constraint is {a, c, b}, and it turns out that this violated constraint is the comb constraint illustrated in Figure 1.2.

We have not given all the details for how one obtains the violated DP-constraint from the cycle in $\underline{M}^*$ we find in Step 3 of the separation algorithm. We give those details in the next chapter. Also, in Chapter 4, we show how this algorithm can be expanded to include current solutions x^* that do not satisfy certain properties.

Chapter 3

Translating a Cycle into a DP-constraint

In Chapter 2, we discussed the details of the Letchford separation algorithm for DP-constraints as presented in [LE00]. Recall that at the end of the algorithm, we form the graph $\underline{M}^*$, which consists of the red edges (the edges of the planar dual of the support graph $G^* = (V^*, E^*)$) and the green edges (which represent the usable dominoes). If we find a cycle in $\underline{M}^*$ of cost less than 1 and containing an odd number of green edges, then by the results in [LE00] this cycle represents a violated DP-constraint.

Although it is clear from the discussions in [LE00] that the cycle found represents a violated DP-constraint, the exact details of how to efficiently find that DP-constraint are not included in the paper. In this chapter, we describe explicitly how this can be accomplished. Furthermore, we describe a process for finding a "cleaner", equivalent form for a violated DP-constraint that CONCORDE, the package we used for testing, is much more likely to accept as a cutting plane. As a result of using this process, the inequalities found as cutting planes during our testing were used more efficiently by CONCORDE. In fact, in some cases we were able to take inequalities that were previously rejected by CONCORDE (due to the size of the formulation) and change them into an acceptable form.

Recall from Chapter 2 that the DP-constraint made up of p dominoes is written as:

$$\sum_{j=1}^{p} x(\delta(D_j)) + \sum_{j=1}^{p} x(E(A_j:B_j)) + x(F) \geq 3p + 1 .$$

For each of the p dominoes, let $C_j = V \setminus (A_j \cup B_j)$ and let $E(A_j:B_j:C_j) = E(A_j:B_j) \cup E(B_j:C_j) \cup E(A_j:C_j)$. Then, the reader can verify that the DP-constraint can also be represented as:

$$\sum_{j=1}^{p} x(E(A_j:B_j:C_j)) + x(F) \geq 3p + 1 . \qquad (3.1)$$

As explained in Chapter 2, each green edge in a cycle of length less than l in $\underline{M}^*$ corresponds to a domino. The translation of such a cycle into a DP-constraint consists of finding the two terms on the left-hand side of (3.1); the calculation of the right-hand side is trivial, as p is simply the number of green edges in the cycle.

Calculating $\sum_{j=1}^{p} x(E(A_j:B_j:C_j))$ is straightforward. For each green edge j in our cycle, the vertex sets A_j, B_j and C_j are found by first generating the three corresponding edge-disjoint shortest paths of minimum total weight; the algorithm for generating these paths is described in Chapter 4. Once we have the three paths, a copy of G^* is created and the edges in the copy crossed by the paths are removed from the copy. By Lemma 5 in [LE00], deleting these edges results in three separate connected components, each with its own vertex set. These components, and thus vertex sets, are found by applying a depth-first search algorithm to our copy of G^*. In [C01], it is proved that any two of these three sets can be chosen as A_j and B_j with respect to a domino D_j (i.e. any two can be chosen as the "inside" of the domino). Each of the three possible choices of two sets results in three equivalent DP-

constraints (although the handles are different in each). For consistency, we assign the two smallest vertex sets as representatives for A_j and B_j.

Finding F is slightly more complicated. Recall from Chapter 2 that we can obtain F if we know the handle $H \subset V$. Let R^*, which may be empty, be the edges in G^* which are crossed by the red edges in the cycle in $\underline{M}^*$. From Lemma 3 in [LE00], the edge subsets $E^*(A_1:B_1)$, ..., $E^*(A_p:B_p)$, R^* support a cut in G^* and the set of vertices for this cut represent the handle. For consistency, we assign H to be the vertex set in G^* on the side of the cut with more vertices. Since $V^* = V$, H is also the handle in our original, complete graph $K_n = (V, E)$.

To obtain H, we need another copy of G^*. We remove the edges e occurring an odd number of times in the subsets $E^*(A_1:B_1)$,..., $E^*(A_p:B_p)$, R^* from the copy of G^*. This breaks the graph G^* into at least two connected components, as can be seen by drawing a cut in a connected graph and removing the edges in the graph crossing the cut. Moreover, each edge removed from the copy of G^* has exactly one end in H. Using a depth-first search strategy, we process the list of deleted edges, placing one end vertex, as well as all other vertices in the copy still connected to this vertex, on one side of the cut. We place the other end vertex and all adjacent vertices on the other side of the cut. Continuing in this manner until all the vertices have been examined, we form H.

For each i = 1, 2, ...,p, let E_i be the set of edges in $E(A_i:B_i)$. As discussed in Chapter 2, there is a unique $F \subseteq E$ such that the edge subsets E_1, ..., E_p, F support the cut $\delta(H)$ in K_n.

Let $F = F_1 \cup F_2$. If $e \in E$ is in $\delta(H)$ in K_n and also in an even number (including 0) of the sets $E_1,\ldots, E_p$, it should be in the cut and so we add it to the set F_1, making μ_e odd for the edge subsets $E_1, \ldots, E_p, F$. If $e \notin \delta(H)$, but is in an odd number of the sets $E_1, \ldots, E_p$, we do not want it included in the cut, so we add it to F_2, making μ_e even. Enumerating the edges in F gives x(F).

Now, we have both terms from the left-hand side of (3.1) and thus everything we need for our DP-constraint. However, the inequality is not exactly in the right form for us. The testing software package CONCORDE requires that all inequalities added as cutting planes be in the closed set form

$$\sum_{i=1}^{k} x(\delta(L_i)) \geq \varphi$$

for vertex subsets $L_1, \ldots, L_k$, $L_i \neq L_j$, $i \neq j$, $\forall\ i, j \leq k$.

The following definition is needed for the ensuing discussion on how to write constraints in the (3.1) form as constraints in this closed set form for CONCORDE.

Definition: A domino D_j, $1 \leq j \leq p$, is *ordinary* if

a) $E(A_j:B_j) \subset \delta(H)$;

b) for each edge $e \in E(A_j:B_j)$, $e \notin E(A_i:B_i)$ for any other dominoes D_i, $i \neq j$, $i = 1, \ldots, p$.

If a domino D_j, $1 \leq j \leq p$, is not ordinary, it is called *non-ordinary*.

Since $F = F_1 \cup F_2$, (3.1) can be written as:

$$\sum_{j=1}^{p} x(E(A_j:B_j:C_j)) + x(F_1) + x(F_2) \geq 3p + 1 . \qquad (3.2)$$

For the ordinary dominoes D_j, we use the following substitution in (3.2):

$$x(E(A_j:B_j:C_j)) = x(\delta(D_j)) + x(E(A_j:B_j)) .$$

For the non-ordinary dominoes D_j, we use:

$$x(E(A_j:B_j:C_j)) = x(\delta(A_j)) + x(\delta(B_j)) - x(E(A_j:B_j)) .$$

These two substitutions allow us to write (3.2) as:

$$\sum_{\substack{D_j \\ ORDINARY}} x(\delta(D_j)) + \sum_{\substack{D_j \\ ORDINARY}} x(E(A_j:B_j)) + \sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(\delta(A_j)) + \sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(\delta(B_j)) -$$

$$\sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(E(A_j:B_j)) + x(F_1) + x(F_2) \geq 3p + 1 . \qquad (3.3)$$

Lemma 3.1. The edges in $\delta(H)$ can be partitioned into three different sets:

a) those in $E(A_j:B_j)$ for D_j ordinary, $j = 1, 2, \ldots, p$;

b) those in F_1;

c) those in an odd number of the edge subsets $E(A_j:B_j)$ for D_j non-ordinary, $j = 1, 2, \ldots, p$; call this set of edge subsets $\mathcal{9}$.

Proof: Statement a) follows directly from the definition of ordinary, as $\mu_e = 1$ for these edges. Statement b) follows from the description of the set $F = F_1 \cup F_2$. Statement c) refers to the edges not accounted for in a) and b). □

Rearranging (3.3) and adding and subtracting $x(\mathcal{9})$, we get:

$$\sum_{\substack{D_j \\ ORDINARY}} x(\delta(D_j)) + \sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(\delta(A_j)) + \sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(\delta(B_j)) + \sum_{\substack{D_j \\ ORDINARY}} x(E(A_j:B_j)) +$$

$$x(F_1) + x(\vartheta) - (\sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(E(A_j:B_j)) + x(\vartheta) - x(F_2)) \geq 3p + 1. \qquad (3.4)$$

By Lemma 3.1,

$$x(\delta(H)) = \sum_{\substack{D_j \\ ORDINARY}} x(E(A_j:B_j)) + x(F_1) + x(\vartheta) .$$

Substituting into (3.4) gives:

$$\sum_{\substack{D_j \\ ORDINARY}} x(\delta(D_j)) + \sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(\delta(A_j)) + \sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(\delta(B_j)) + x(\delta(H)) -$$

$$(\sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(E(A_j:B_j)) + x(\vartheta) - x(F_2)) \geq 3p + 1 . \qquad (3.5)$$

The DP-constraint (3.5) is almost in the closed set form required by CONCORDE, except for the term:

$$- (\sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(E(A_j:B_j)) + x(\vartheta) - x(F_2)). \qquad (3.6)$$

For reasons soon to become apparent, we need the following lemma.

Lemma 3.2. For each edge $e \in E$, the coefficient of x_e in (3.6) is always non-positive and even.

Proof: Clearly for any edge e not in ϑ, F_2 or $E(A_j:B_j)$ for a non-ordinary domino D_j, the coefficient of x_e equals 0 in (3.6). Now, for all D_v non-ordinary, v = 1, 2, ..., p, let the number of subsets $E(A_v:B_v)$ that e appears in be Q_e for each $e \in E$. The edges that are in ϑ are those in $\delta(H)$ and for which Q_e is odd. The edges that are in F_2 are those not in $\delta(H)$ and for which Q_e is odd. If Q_e is even, then x_e's coefficient in (3.6) is $-(Q_e + 0 + 0)$, which is either 0 or an even negative integer. If Q_e is odd and $e \in \delta(H)$, the coefficient of x_e is $-(Q_e + 1)$. If Q_e is odd and $e \notin \delta(H)$, x_e's coefficient is $-(Q_e - 1)$. Thus, in both cases where Q_e could be odd, the coefficient of x_e is either 0 or an even negative integer. The result follows.□

Given the previous lemma, we are now prepared to state the key result of this chapter. Credit is due to Adam Letchford [AL00] for pointing us in the right direction.

For each edge e for which x_e has a non-zero coefficient $-2k_e$ in (3.6), $k_e \in \{1, 2, ...\}$, replacing $-2k_e x_e$ with $k_e(x(\delta(e)))$ in (3.6) and adding $4k_e$ to the right-hand side of (3.5) gives the required closed set form for CONCORDE. To see this, let e = ab. If we add together the degree equations (1.2) for vertices a and b, we obtain $x(\delta(a)) + x(\delta(b)) = 4$. We can rewrite this as $x(\delta(e)) + 2x_e = 4$. Rearranging gives $-2x_e = x(\delta(e)) - 4$. From Lemma 3.2, the coefficient of x_e in (3.6) is some non-negative multiple of -2. It follows that if the coefficient for x_e in (3.6) is $-2k_e$, we can replace it with $k_e(x(\delta(e))) - 4k_e$. Moving $4k_e$ to the right-hand side of (3.5) gives our final closed set form for the DP-constraint, which is equivalent to the DP-constraint (3.1):

$$\sum_{\substack{D_j \\ ORDINARY}} x(\delta(D_j)) + \sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(\delta(A_j)) + \sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(\delta(B_j)) + x(\delta(H)) +$$

$$\sum_{e \in E} k_e(x(\delta(e))) \geq 3p + 1 + 4\sum_{e \in E} k_e \,. \qquad (3.7)$$

Despite the fact that (3.7) is in the closed set form, there is one drawback. Making the substitution just presented may result in a large number of vertex sets of size 2 in our final closed set form. Since CONCORDE's internal format requires that each of the vertex sets in (3.7) appear on a separate line in a file, this file can get quite long and slow down the execution of the program. Moreover, if the number of vertex sets is too large, CONCORDE will not accept the inequality at all. To reduce the number of vertex sets in (3.7) of size 2 and thus enable CONCORDE to work more efficiently and accept more of our inequalities, the following action is taken.

Recall from earlier in this chapter that for our current dominoes $D_j = A_j \cup B_j$, $j = 1,\ldots, p$, the two vertex sets A_j and B_j were arbitrarily chosen to be the smallest of the three sets A_j, B_j and C_j, and that it is shown in [C01] that choosing a different pair of these sets to be the domino results in an equivalent DP-constraint (with a different handle). However, our choice of A_j and B_j as the two vertex sets constituting the domino may not have been the wisest. In certain cases, choosing a different pair of the sets A_j, B_j and C_j can change the status of a domino, in terms of being non-ordinary or ordinary.

Consider the domino-comb $\{H; D_1, D_2, D_3\}$ in K_8 with handle $H = \{2, 4, 6, 7\}$, and dominoes $D_1 = \{1\} \cup \{2\}$, $D_2 = \{3\} \cup \{4\}$ and $D_3 = \{5\} \cup \{6\}$, shown in Figure 3.1(a), and note that in this particular case, the domino-comb is also a comb. If we replace D_1 with $D_1' = A_1 \cup C_1$, we obtain an equivalent domino-comb with a different handle, shown in Figure 3.1(b). Similarly, if we replace D_1 with $D_1'' = B_1 \cup C_1$, we obtain another equivalent domino-comb, again with a different handle, as shown in Figure 3.1(c). Notice that the domino-comb in (a) has three ordinary dominoes, while the equivalent domino-combs in (b) and (c) each have two ordinary dominoes; D_1' and D_1'' are both non-ordinary.

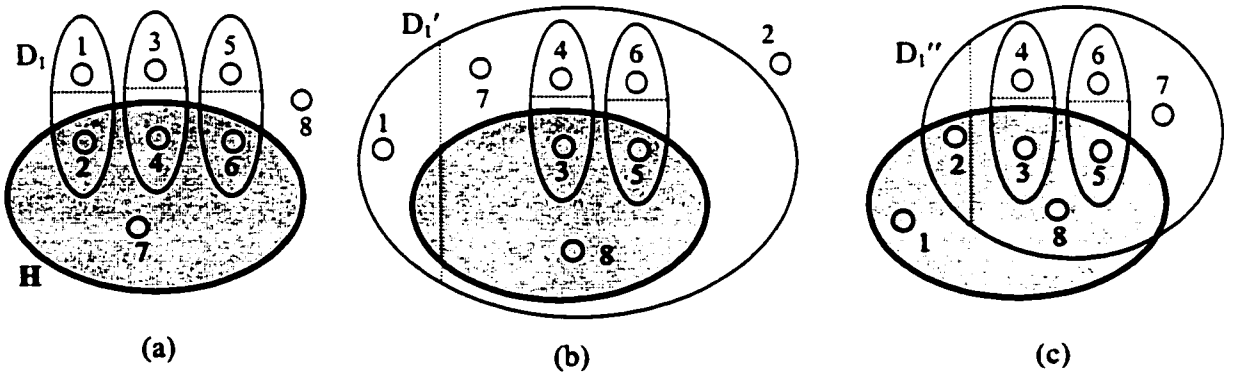


Figure 3.1: Three equivalent domino-combs

To improve our closed set form (3.7) for CONCORDE, we need to decide for each domino which two of the vertex sets A, B and C should be included in the domino in order to increase the number of ordinary dominoes. For a DP-constraint with p dominoes, there are 3^p possible domino-combs obtainable from *compartment switching* (i.e. choosing a different pair of the vertex sets A, B and C for the domino). Thus, trying each possible combination and searching for the one with the maximum number of ordinary dominoes is futile. In addition, compartment switching in an iterative manner is not trivial, as doing this for one

domino D_j could possibly change the status of another domino D_i from ordinary to non-ordinary. We deal with this problem as discussed next.

It turns out that the edge subsets $E(A_1:B_1:C_1)$, ..., $E(A_p:B_p:C_p)$, F also support a cut $\delta(H'')$ in K_n for some vertex subset $H'' \subset V$ [C01]. Let μ_e'' be the multiplicity of edge $e \in E$ in this set. Then, H'' is the handle formed from the edges with μ_e'' odd. The μ_e'' values are the coefficients of the x_e's in (3.1), which we have already calculated. To check if we can change the status of a domino from non-ordinary to ordinary, we use the following definition, found in [C01].

Definition: A domino D_j, j =1,... ,p, is *regular* if and only if $\mu_e'' = 1$ for all edges $e \in E(A_j:B_j)$. Equivalently, D_j is regular if and only if it is ordinary and $E(A_j:B_j) \cap \delta(D_i) = \varnothing$ for all other dominoes D_i, $i \neq j$, i = 1, ..., p. This implies that a regular domino is also ordinary.

Note that if all the dominoes for a DP-constraint are regular and $p \geq 3$, then the constraint is a comb inequality [C01]. Also note that it follows from the definition of regular that switching the compartments of a domino D_j does not affect the "regular" status of another domino D_i, as it may in the case of ordinary. Thus, we can increase the number of ordinary dominoes for a DP-constraint by maximizing the number of regular dominoes using the following method.

If all our dominoes are ordinary, this means that the coefficient of x_e in (3.6) is 0 for all $e \in E$ and we can bypass this process. If this is not the case, then for each non-regular domino $D_j = A_j \cup B_j$, if $\mu_e'' = 1$ for all edges $e \in E(A_j:C_j)$, then replace B_j in D_j with C_j and re-classify D_j as regular. Otherwise, check if $\mu_e'' = 1$ for all edges $e \in E(B_j:C_j)$, and if so, replace A_j with C_j and re-classify the domino as regular. After processing all the non-regular dominoes, we will have maximized the number of regular dominoes for our DP-constraint. Of course, all these regular dominoes are ordinary as well. As a last step, we can now check the remaining non-regular dominoes to see if they are ordinary. Note that if after checking all the non-regular dominoes, none have been changed to regular, our handle H remains the same. Otherwise, if any of our non-regular dominoes have been re-classified as regular, H must be re-calculated using the new domino(es).

In the end, this leaves us with definitive classifications of our dominoes as ordinary or non-ordinary. Having used the method just described to maximize the number of regular dominoes, and as a result decrease the number of vertex sets of size 2 in (3.7), the final DP-constraint can be written to a file for use by CONCORDE.

Chapter 4

Implementation & Enhancements

This chapter contains a detailed description of our implementation of Adam Letchford's algorithm for finding violated DP-constraints. The program is called DP-CUTFINDER and contains roughly 5000 lines of code. Note that the code was written in ANSI C and compiled using the GNU C compiler (version 2.7.2).

Several enhancements that we made to the algorithm are also discussed in this chapter. In particular, we discuss a method by which the Letchford algorithm can be used to sometimes find violated DP-constraints even in situations where the support graph of x^* is non-planar. It should be very clear in the text which ideas are Letchford's and which are ours.

4.1 Implementation Details

We start with a description of the DP-CUTFINDER implementation. Unless otherwise noted, we assume that the support graph $G^* = (V^*, E^*)$ of our current solution vector x^* is planar. Dealing with non-planar support graphs is discussed in Section 4.2.

Step 1: Find the red edges

We find the red edges by finding the edges of the planar dual $\underline{G}^* = (\underline{F}^*, \underline{R}^*)$ of G^* in a manner that is not the most efficient, but is effective nonetheless. To expedite the testing process, we opted to use a simple algorithm which we developed. We could not find a

description of this algorithm anywhere, but it is very straight-forward and thus unlikely to be new (note that some of the ideas are similar to those described in [K97]). Essentially, we find the faces of G* and this allows us to determine for each edge $e \in E^*$ the two faces of G* which contain e. Then, an edge in $\underline{G}^*$ of weight x_e is formed between the two faces. Doing this for all $e \in E^*$, we create the set of red edges $\underline{R}^*$. It is important to note that it is sometimes possible in general planar graphs to have an edge e which belongs to exactly one face of a planar graph rather than two distinct faces, but this only occurs when e is a cut edge of the graph. Thus this never happens in any of our support graphs G*, as they contain no cut edges.

To find the faces of G*, we first create a new directed graph $\vec{G} = (\vec{V}, \vec{E})$ from G* by replacing each undirected edge ij by two arcs (i, j) and (j, i). Traversals of these arcs are marked in a list of size $|\vec{E}|$. We introduce a dummy vertex z, which gives us a starting point and allows us to easily find the outer face. We ensure that z lies in the outer face of G* by assigning z an x-coordinate that is less than that of the vertex u_0 in G* with the smallest x-coordinate and a y-coordinate that is equal to that of the same vertex. Note that angle calculations are always made in a counterclockwise direction (see Appendix A for details). The algorithm is as follows:

considering all vertices which are the head of an arc in $\vec{E}$ with tail u_0, find the vertex u_1 for which angle $\angle z u_0 u_1$ is smallest; {begin find outer face}

let (u_0, u_1) be the current arc;

repeat

represent the current arc by (u_j, u_k);

add arc (u_j, u_k) to the outer face;

mark arc (u_j, u_k);

considering all vertices which are the head of an arc in $\bar{E}$ with tail u_k, find the vertex u_p for which angle $\angle u_j u_k u_p$ is smallest;

let (u_k, u_p) be the current arc;

until current arc = (u_0, u_1); {end find outer face}

while (there is an unmarked arc (a, b) in $\bar{G}$)

let (a, b) be the current arc;

repeat {begin find next face F}

let (j, k) represent the current arc;

add arc (j, k) to face F;

mark arc (j, k);

considering all vertices which are the head of an arc in $\bar{E}$ with tail k, find the vertex p for which angle $\angle$jkp is smallest;

let (k, p) be the new current arc;

until current arc = (a, b); {end find next face F}

end while;

We now go through a couple of iterations of the algorithm on our small example from before; its directed graph is shown in Figure 4.1.

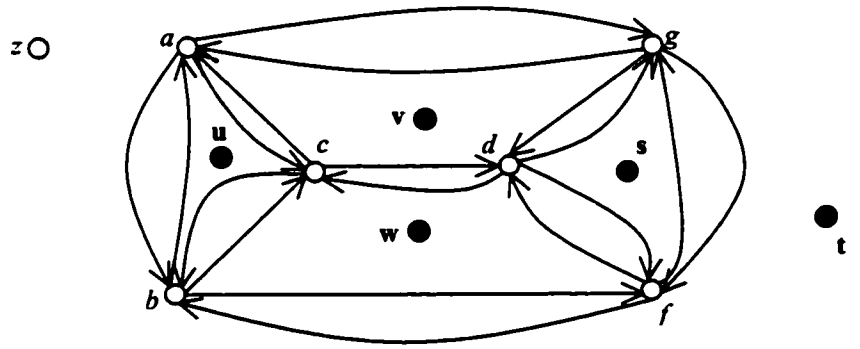


Figure 4.1: Planar dual example

To find the outer face **t**, we begin by calculating the smallest of the angles $\angle zab$, $\angle zac$ and $\angle zag$. Since $\angle zab$ is smallest, we mark the arc *(a, b)* as visited and add it to face **t**. The current arc is now *(a, b)*. Compare $\angle abc$ and $\angle abf$. Since the latter is smaller, mark the arc *(b, f)* and add it to face **t**. Move to *(b, f)* as the current arc and compare $\angle bfd$ and $\angle bfg$. Since $\angle bfg$ is smaller, mark *(f, g)* and add it to face **t**. Go on to *(f, g)*, compare angles, mark arc *(g, a)* and add it to face **t**. Move to *(g, a)* and compare $\angle gab$ and $\angle gac$. Since $\angle gab$ is smallest and arc *(a, b)* is already in **t**, this face is complete.

To find face **u**, we let the unmarked arc *(a, c)* be our current arc and we add it to face **u**. Compare $\angle acb$ and $\angle acd$. Since the first angle is smaller, mark arc *(c, b)* and add it to face **u**. Move to the current arc *(c, b)* and compare $\angle cba$ and $\angle cbf$. Since $\angle cba$ is smaller, mark arc *(b, a)* and add it to face **u**. Compare $\angle bac$ to $\angle bag$. Since the first angle is smaller and

(a, c) is already in **u**, we find the next unmarked arc *(a, g)* and start computing **v**. We continue in this manner until all arcs in $\vec{G}$ are visited and marked.

Step 1 Complexity: For the planar graph G*, let the number of vertices, edges and faces be n, m and f, respectively. Then, $m \in \Theta(n)$ and $f \in O(n)$; please see Appendix B for details. Initializing the necessary data structures and determining the initial starting vertex are $O(n^2)$ and $\Theta(n)$, respectively. The angle calculations are $\Theta(1)$. During the algorithm, an arc (i, j) of $\vec{G}$ becomes current and has all of its angles $\angle ijp$, for arc (j, p) in $\vec{G}$, calculated exactly once. Letting o_j represent the out degree of vertex j in $\vec{G}$, we have o_j angle calculations. We have to repeat these o_j angle calculations for a vertex j a total of o_j times, since there are o_j arcs of the form (i, j). So in all, we need to do at most $\sum_{j=1}^{n} (o_j)^2$ calculations. Noting that $\sum_{j=1}^{n} (o_j)^2 \le (\sum_{j=1}^{n} o_j)^2$, and that $\sum_{j=1}^{n} o_j = |\vec{E}| = 2|E^*|$, which is $\Theta(n)$ since G* is planar (see Appendix B), this gives a complexity of $O(n^2)$ for all the angle calculations, and thus a total complexity of is $O(n^2)$ for the entire planar dual algorithm. As mentioned earlier, this is not the most efficient algorithm for this problem, but is certainly fast enough for our purposes; for our largest TSP test problems, the routine took no more than a couple of seconds to run.

Step 2: Find the green edges

For each vertex pair {s, t} in $\underline{G}^*$, we must find three edge-disjoint s to t paths of minimum total weight, w_{st}. If $w_{st} < 4$, we save the edge and associate with it a weight of $w_{st} - 3$. Each saved edge is a green edge. The method we used to find the three paths is described next.

We transform this problem into a minimum cost flow problem and apply the Successive Shortest Path Algorithm as outlined in [AMO93]. If the reader is not familiar with the minimum cost flow problem, we suggest [AMO93] as a reference. Since this algorithm requires the graph we are working with to be directed, we first transform $\underline{G}^*$ into a digraph D, represented as an adjacency list. To do this, replace each edge ij in $\underline{G}^*$ with two arcs (i, j) and (j, i) in D, each with weight x_{ij}. Note that in [AMO93], it is proved that solving the flow problem in D also solves the problem in $\underline{G}^*$. Assign each arc (i, j) in D a capacity equal to the number of edges between vertices i and j in $\underline{G}^*$, in order to ensure that the paths we find using the flow are edge-disjoint. The problem is to find a minimum cost s to t flow in D of value 3 for each vertex pair {s, t} in $\underline{G}^*$.

Several data structures are needed. There is a matrix C, which holds the weights (or costs) of all arcs in D. Note that the cost of arc (i, j) is the same as the cost of arc (j, i), and all costs are positive. In a matrix X, the flows on D's arcs are stored; at different points in the algorithm, the flow on arc (i, j) is not necessarily the same as the flow on arc (j, i). At each stage of the algorithm we have an auxiliary digraph A, which has a directed s to t path for every flow-augmenting path from s to t in D. The digraph A does not need to be stored

explicitly (since it is built from information stored in C, D and X). We do the following for each vertex pair {s, t} in $\underline{G}^*$. Please refer to Appendix C for a detailed example.

(2a) Initialization –

X[i, j] := 0 for all arcs ij in D;

y_i := 0 for all vertices i in D (this is called the *potential*);

FV := 0 (this is the total flow value; we are done when FV = 3);

w_{st} := 0 (this is the minimum cost value of the three edge-disjoint s to t paths;

we can stop if $w_{st} \geq 4$);

(2b) We form the auxiliary digraph, A, from D. This graph consists of the original digraph D, but for any arc (i, j) for which the flow X[i, j] is greater than 0, we add the reverse arc (j, i) with cost -C[i, j]. If X[i, j] = capacity, then we remove arc (i, j). Now the flow-augmenting paths are in one-to-one correspondence with the s to t directed paths in A. As previously mentioned, we do not need to actually create and store A. Instead we use the procedure described next which, given an arc (i, j), determines if it exists in A, and what its cost c_{ij} is in A. In this procedure we exploit the fact that our flows will only be non-zero in at most one of the arcs (i, j) and (j, i) in D. We also exploit the fact that if we have added a reverse arc for (j, i) in A, the result is two arcs (i, j) in A, one with cost C[i, j] and the other with cost -C[i, j] (recall C[i, j] = C[j, i]). However, because in any shortest path (or minimum cost) algorithm the arc with the negative cost will be chosen, in building A there is no need to consider the arc with positive cost.

```
procedure check_arc_ij (i, j: indices; var exists: boolean; var c_ij: real);
begin
        exists := true;
        if X[i, j] = capacity then exists := false;
        else if X[j, i] > 0 then c_ij := -C[i, j];   (reverse arc)
            else c_ij := C[i, j];                    (forward arc)
end;
```

(2c) We find a shortest s to t dipath in A. To do this as efficiently as possible, we wish to use the heap implementation of Dijkstra's Shortest Path Algorithm, described in [AMO93]. However, in order to do so, there cannot be any negative costs. Consequently, we use the vertex potentials y_v to calculate the following set of reduced costs as input to the Dijkstra algorithm (these ideas are described briefly in [AMO93]):

for each arc (i, j) in A: $c^y_{ij} := c_{ij} - y_i + y_j$ (c_{ij} as calculated in check_arc_ij)

Note that we stop the algorithm once we reach vertex t, so not all vertices will necessarily be reached (and receive permanent labels). Some information must also be saved while finding the shortest dipath:

T: set of vertices that were reached and received permanent labels

d_i: for each vertex i in T, d_i is the length of shortest s to i dipath in A (using reduced costs)

P: shortest s to t dipath found

(2d) Recall that the s to t dipath P found in A in (2c) corresponds to a minimum cost flow-augmenting path in D. Calculate ε, the amount by which we can augment the flow using this flow-augmenting path, in the usual way (except if $\varepsilon + FV > 3$, then $\varepsilon = 3 - FV$). For each arc (i, j) in P, add ε to the flow or subtract ε from the flow, depending on whether (i, j) was a forward arc or a reverse arc in A:

if $X[j, i] > 0$, then $X[j, i] := X[j, i] - \varepsilon$; (reverse arc)

else $X[i, j] := X[i, j] + \varepsilon$; (forward arc)

(2e) Update:

$FV := FV + \varepsilon$;

cost of dipath $P := d_t + y_s - y_t$; (again making use of the potentials to efficiently get the cost of P using the original costs)

$w_{st} := w_{st}$ + cost of dipath P

(2f) Check:

if $w_{st} \geq 4$, stop; there is no green edge between vertices s and t

if $FV = 3$, stop; the three edge-disjoint s to t dipaths of minimum total weight have been found: they correspond to the arcs in D with flow value > 0

otherwise if $FV < 3$, update the y values and go back to (2b) - again, as briefly outlined in [AMO93], we can do this as follows:

$y_i := y_i - d_i + d_t$ (if vertex i got permanently labelled)

$y_i := y_i$ (if vertex i did not get permanently labelled)

Step 2 Complexity: Let f be the number of faces in G^*. Then, for each of the f(f-1)/2 vertex pairs {s, t} in $\underline{G}^*$, the shortest path routine may be called three times. Since we use the heap implementation of Dijkstra's Shortest Path Algorithm, which is $\Theta(f \lg f)$, the total complexity is $\Theta(f^3 \lg f)$. Since $f \in O(n)$, as shown in Appendix B, this step is $O(n^3 \lg n)$. Note that in [LE00] a paper describing a linear time shortest path algorithm for planar graphs [HKRS97] is suggested for this step. Clearly, this would reduce the complexity of Step 2 to $O(n^3)$. However, this algorithm is extremely complicated and would be very long and difficult to implement. Thus, because of time constraints, it was not implemented.

Step 3: Find cycles in $\underline{M}^*$

We define the graph $\underline{M}^*$ with vertex set $\underline{F}^*$ (the dual vertices) and an edge set consisting of all the red and all the green edges, found in Step 1 and Step 2 respectively. In [LE00], the next step is to find in $\underline{M}^*$ a cycle of minimum weight containing an odd number of green edges and any number of red edges. If this cycle has weight less than 1, it corresponds to a violated DP-constraint. Otherwise, there are no DP-constraints violated by x^*, and we can stop running DP-CUTFINDER. In our work, the algorithm is taken one step further. Based on the research in [BCCN96], as well as our own preliminary testing, it seems to be advantageous in the cutting plane method to add a number of violated constraints to our LP which cut off the current solution x^* rather than just a single constraint. So we decided that in the case that there is a cycle in $\underline{M}^*$ with an odd number of green edges and weight less

than 1, that we would try to find as many such cycles as possible, rather than just the minimum weight one. All these cycles correspond to violated DP-constraints which can be added to our LP. The thinking behind this is that the more DP-constraints we generate for CONCORDE, the faster we will (hopefully) reach an optimal solution.

We first transform $\underline{M}^*$ so that we can look for a minimum weight cycle with the total number of edges being odd (as opposed to only the number of green edges being odd). For every red edge ij of weight K, we split the red edge into two edges of equal weight K/2. This guarantees that there are an even number of red edges in any cycle in $\underline{M}^*$, thus ensuring that any odd-length cycle in $\underline{M}^*$ has an odd number of green edges. This first transformation, shown in Figure 4.2, results in a new intermediate vertex r for every red edge.

Figure 4.2: Transformation of $\underline{M}^*$ to $\underline{I}^*$

At this point, the graph, $\underline{I}^*$, consists of $\underline{F}^*$ and the new set of intermediate vertices, as well as the new red edge set and the green edge set. Now in order to find an odd-length cycle of minimum weight in $\underline{I}^*$, another transformation is required. We do as in [GLS88] and create a bipartite graph $\underline{B}^*$ from $\underline{I}^*$ in the following way. For each vertex in $\underline{I}^*$, create two vertices v_1 and v_2 in $\underline{B}^*$. If v and w are joined in $\underline{I}^*$ and the edge has cost c, join v_1 and w_2 in $\underline{B}^*$, as well as v_2 and w_1 and give these edges cost c. This process is depicted in Figure 4.3.

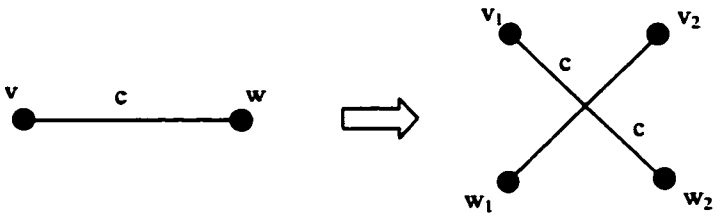


Figure 4.3: Transformation of $\underline{I}^*$ to $\underline{B}^*$

Now, since $\underline{B}^*$ is bipartite, any path from v_1 to v_2 is odd in edge length. In summary, for all vertices v in $\underline{I}^*$, the minimum weight odd cycle containing v is found by making the transformation to $\underline{B}^*$ and solving the shortest path problem from v_1 to v_2 (i.e. we solve a shortest path problem for each vertex in $\underline{I}^*$). Once again, the heap implementation of Dijkstra's Algorithm is used. If any of these paths found have cost less than 1, they correspond to a violated DP-constraint. Note that to find the DP-constraint which violates x^* the most, we would find the minimum cost path of all the paths we found. However, as mentioned before, we instead choose to keep all the violated DP-constraints we find at this step to add to our LP.

Although the translation of a path in $\underline{B}^*$ to the cycle we seek in $\underline{M}^*$ is straightforward, we must be careful not to process duplicate cycles. Recall that one of the ways in which our enhanced separation algorithm for DP-constraints differs from Letchford's original algorithm is that we find more than one cycle in $\underline{M}^*$ (of cost less than 1) with an odd number of green edges and any number of red edges. For all pairs $\{v_1, v_2\}$, after using the method just described to find the shortest v_1 to v_2 path of cost less than 1 in $\underline{B}^*$(if one exists), we make the simple translation to a cycle in $\underline{M}^*$. However, before using the technique described in

Chapter 3 to produce violated DP-constraints from these cycles, we must remove any duplicate cycles that may have been generated.

To check if a cycle C1 has a duplicate, we scan through the second cycle C2 with which we are comparing it in search of the first entry in C1's adjacency list (C1[0]). If it's not there, C1 and C2 are not duplicates. Otherwise, if C1[0] = C2[i], we check for C1[1] in positions i-1 and i+1 in C2. If C1[1] is not in either position in C2, C1 and C2 are not duplicates. Otherwise, we compare the remaining elements in C1 and C2, until we are done or we find elements that do not match. Note that each cycle needs to be compared against all other cycles.

Step 3 Complexity: The transformation of $\underline{M}^*$ into $\underline{B}^*$ is $O(n^2)$; the latter structure is represented internally as a matrix. For each vertex v_1 in $\underline{B}^*$, the shortest path to the corresponding v_2 vertex has to be found. Recall that the heap implementation of Dijkstra's Algorithm is $\Theta(k \lg k)$ for a graph with k vertices. Assuming that the number of edges in $G^* \in \Theta(n)$ and the number of faces in $G^* \in O(n)$, the number of vertices in $\underline{B}^*$ is O(n). Since our cycles are stored as doubly-linked adjacency lists, checking for duplicate cycles is $\Theta((\#\text{ of cycles})^2 \times n)$ and the number of cycles is at most O(n). Thus, the total complexity of this step is $O(n^3)$.

Step 4: Translate cycles in $\underline{M}^*$ into DP-constraints for CONCORDE

The steps required for the translation of a cycle in $\underline{M}^*$ to a DP-constraint are described in Chapter 3. To review, we do the following for each cycle found in Step 3:

(4a) Copy the contents of the $|\underline{F}^*|$ by $|\underline{F}^*|$ matrix, holding the red edges values into a digraph.

(4b) For each green edge j in the cycle:

(4b1) using the (adapted) Successive Shortest Path Algorithm, again generate the three corresponding edge-disjoint shortest paths

(4b2) create a copy of G* and remove the edges crossed by the three paths

(4b3) apply a depth-first search algorithm, giving A_j, B_j and C_j, thus finding the domino D_j corresponding to the green edge j

(4c) For each red edge in the cycle, we have to add the corresponding edge in E* to R*, defined in Chapter 2.

(4d) Calculate the handle H:

(4d1) make another copy of G* and remove the edges e with odd multiplicity μ_e in the edge subsets $E^*(A_1:B_1)$, ..., $E^*(A_p:B_p)$, R*

(4d2) each deleted edge has exactly one end in H; use a depth-first search strategy to place all the vertices on either side of the cut

(4e) Find $F = F_1 \cup F_2$.

(4f) Classify the dominoes, represented by the green edges, as being ordinary or non-ordinary. If all dominoes are ordinary, skip to Step (4h).

(4g) For each non-regular domino, check if any can be changed to regular dominoes (and thus ordinary):

(4g1) if yes, make all such changes and recalculate the handle

(4g2) if no, do nothing.

(4h) At this point, we have all the information that we need. Recall the DP-constraint (3.5) for j = 1, ..., p:

$$\sum_{\substack{D_j \\ ORDINARY}} x(\delta(D_j)) + \sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(\delta(A_j)) + \sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(\delta(B_j)) + x(\delta(H)) -$$

$$\left(\sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(E(A_j:B_j)) + x(\vartheta) - x(F_2) \right) \geq 3p + 1$$

We have the handle H and the dominoes are stored in a data structure such that for each domino, we have its classification and the two lists A and B. As well, the coefficients of

$$\left(\sum_{\substack{D_j \\ NOT \\ ORDINARY}} x(E(A_j:B_j)) + x(\vartheta) - x(F_2) \right)$$

are stored in an n by n matrix Y. Examining Y and making the necessary adjustments described in Chapter 3, we can write the vertex sets and the right-hand side to the cuts file for CONCORDE in the same order as they appear in (3.5).

Step 4 Complexity: The most expensive steps are (4b) and (4g). In Step (4b), the operation listed as (4b1) is the most expensive, at $\Theta(n \lg n)$. Since we do this for each green edge in

the cycle and there are at most O(n) of these, (4b) is $O(n^2 \lg n)$. As for (4g), (4g1) is clearly the most expensive; it is $O(n^2)$. So, the complexity of (4g) is $O(n^2)$. Therefore, the complexity of Step 4 is $O((\#\text{ of cycles}) \times n^2 \lg n)$, or $O(n^3 \lg n)$. Note that although the number of cycles as well as the number of green edges per cycle are proportional to n, they are in practice much smaller than n.

The complexities of the individual steps are shown in Table 4.1. In the case of finding a single cycle, Step 2 is the most time-consuming and therefore, the total complexity of DP-CUTFINDER is $O(n^3 \lg n)$. Again, note that as stated in [LE00], the algorithm for finding a single violation can be implemented in $O(n^3)$ if the shortest path algorithm of [HKRS97] is used in Step 2.

Step	Complexity
1	$O(n^2)$
2	$O(n^3 \lg n)$
3	$O(n^3)$
4	$O(n^3 \lg n)$

Table 4.1: Algorithm Complexity

4.2 Dealing with Non-planar Support Graphs

As was discussed earlier, Letchford's separation algorithm only works if G* is planar. Currently, we do not have any software that does planarity testing for us. However, we do have a graph viewer that helps us to determine if there are any edge crossings in G*. If there are crossings but G* is still planar, we simply rearrange vertices until we have a *straight-line*

planar embedding (a planar representation consisting only of straight line segments). Note that every planar graph has a straight-line planar embedding [W36]. Our method is clearly not very sophisticated; some ideas for improvement and automating this process are discussed further on in the section.

Despite the planarity requirement, if G* is non-planar, all hope is not lost. The last enhancement we make is invaluable in that it provides a method for finding violated DP-constraints even when G* is non-planar.

Definition: Given a weighted graph G = (V*, E*), *shrinking a vertex set* $U \subset V$ to form a *shrink vertex* u′ refers to the process of:

(a) deleting all edges in E*(U),

(b) deleting all vertices $u \in U$, and replacing them with the single vertex u′,

(c) replacing all edges $uw \in E^*$, $u \in U$, $w \notin U$, with edge u′w,

(d) and replacing any of the resulting multiple edges between vertices u′ and w with a single edge of weight equal to the sum of the weights of the multiple edges.

If we let G′ = (V′, E′) be the new graph obtained, it represents the support graph of a solution vector $x' \in \Re^k$, where k is the number of edges in the complete graph with $|V'|$ vertices. Figure 4.4 shows an example of shrinking a vertex set $U = \{u_1, u_2, u_3, u_4, u_5\}$.

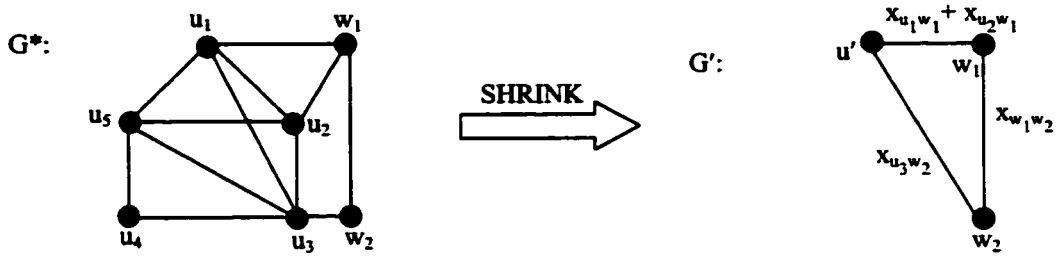


Figure 4.4: A Shrinking Example

Proposition 4.1. If the current LP solution x* induces a support graph G* = (V*, E*) and a set of vertices U ⊂ V can be shrunk in G* to give a new shrink vertex u', solution vector x' and graph G', then for any violated DP-constraint in G' = (V', E'), there is a corresponding violation in G*.

Proof: Suppose we have a DP-constraint a'x ≥ σ' which is violated by x'. Then, from our definition of DP-constraints,

$$\sum_{j=1}^{p} x'(E(A'_j:B'_j:C'_j)) + x'(F') < 3p + 1 .$$

Define the analogous DP-constraint ax ≥ σ for G*. Let $A_j = A'_j$, $B_j = B'_j$ and $C_j = C'_j$, except substitute vertex set U in place of u' in any set in which u' appears. Let F = F', but replace each edge u'v by the edge set E(U:{v}). Furthermore, let the prime variables V', E', H', etc. be to G' as their non-prime counterparts are to G*.

We know that $\{E'_1, \ldots, E'_p, F'\}$ supports δ(H') in $K_{|V'|}$, H' ⊂ V' where E'_i represents the edges in the semicut for domino D_i. Taking into consideration the substitutions for A'_j, B'_j, C'_j and F' and letting $E_i = E(A_i:B_i)$, for each edge wz in K_n, the multiplicity μ_{wz} with respect to the multiset $\{E_1, \ldots, E_p, F\}$ is:

$$\mu_{wz} = \begin{cases} 0, & \text{if } w, z \in U \\ \mu'_{wz}, & \text{if } w, z \notin U \\ \mu'_{wu'}, & \text{if } w \notin U, z \in U \\ \mu'_{u'z}, & \text{if } w \in U, z \notin U \end{cases}$$

WLOG, suppose $u' \in H'$. Let $H = H' \setminus \{u'\} \cup U$. Since $\delta(H') = \{e \in E(K_{|V'|}): \mu'_e \text{ odd}\}$ and from the definition of μ_{wz} just presented, it follows that $\delta(H) = \{e \in E: \mu_e \text{ odd}\}$. Therefore, $\{E_1, \ldots, E_p, F\}$ supports $\delta(H)$ in K_n and $ax \geq \sigma$ is a DP-constraint.

We must also show that $ax \geq \sigma$ cuts off x^*, that is $ax^* < \sigma$. From the set of substitutions just described, for each edge wz in G^*, we have:

$$a_{wz} = \begin{cases} 0, & \text{if } w, z \in U \\ a'_{wz}, & \text{if } w, z \notin U \\ a'_{wu'}, & \text{if } w \notin U, z \in U \\ a'_{u'z}, & \text{if } w \in U, z \notin U \end{cases}$$

In addition, for each edge wz in the shrunk graph:

$$x'_{wz} = \begin{cases} x^*_{wz}, & \text{if } w, z \neq u' \\ \sum_{u \in U} x^*_{uz}, & \text{if } w = u' \\ \sum_{u \in U} x^*_{wu}, & \text{if } z = u' \end{cases}$$

Therefore, from the relationships between a, a′ and x*, x′, ax* = a′x′. Thus, $ax \geq \sigma$ cuts off x*. □

Note that once we shrink a set of vertices in G* to give G′, the corresponding solution vector x′ still satisfies the lower bound and subtour constraints for any $S \subset V$, but may no longer satisfy the degree constraints or the upper bound constraints for SEP(n). However, although it is not first presented as such in [LE00], Letchford does point out in this same paper that the separation algorithm will also work when our solution vector x′ only satisfies the lower bound and subtour constraints for all $S \subset V$, as long as the support graph is planar. This means that if we can find a set of disjoint subsets of vertices in G* to shrink such that G′ is planar, our separation routine will find DP-constraints violated by x′ (if there are any). Then, to find the corresponding DP-constraints violated by x*, we can use the construction presented in Proposition 4.1. In this way, we are able to sometimes find violated DP-constraints for our original solution vector x* even when G* is non-planar.

If we allow multiple shrinkings in G*, it is possible that the resulting graph G′ may contain a cut edge, even though we know that the original graph G* does not have any cut edges or cut nodes. A cut edge ab created in G′, causes problems for our algorithm for finding the faces of G′, as it assumes that the graph is 2-edge connected. To overcome this problem, we can simply shrink the node set {a, b}.

It is also worth mentioning that after shrinking, our algorithm is no longer exact, but rather a heuristic. DP-constraints violated by x* may still exist, but neither DP-CUTFINDER nor

Letchford's original algorithm will find them. One such example is shown in Figure 4.5. The graph G* is clearly non-planar, but there is one violated DP-constraint (actually a comb) with H = {f, g, h}, D_1 = {a, g}, D_2 = {e, f} and D_3 = {c, h}. If we shrink {e, h} in G* to form u', we get G', which is planar. However, it can easily be verified that are no DP-constraints violated by x'.

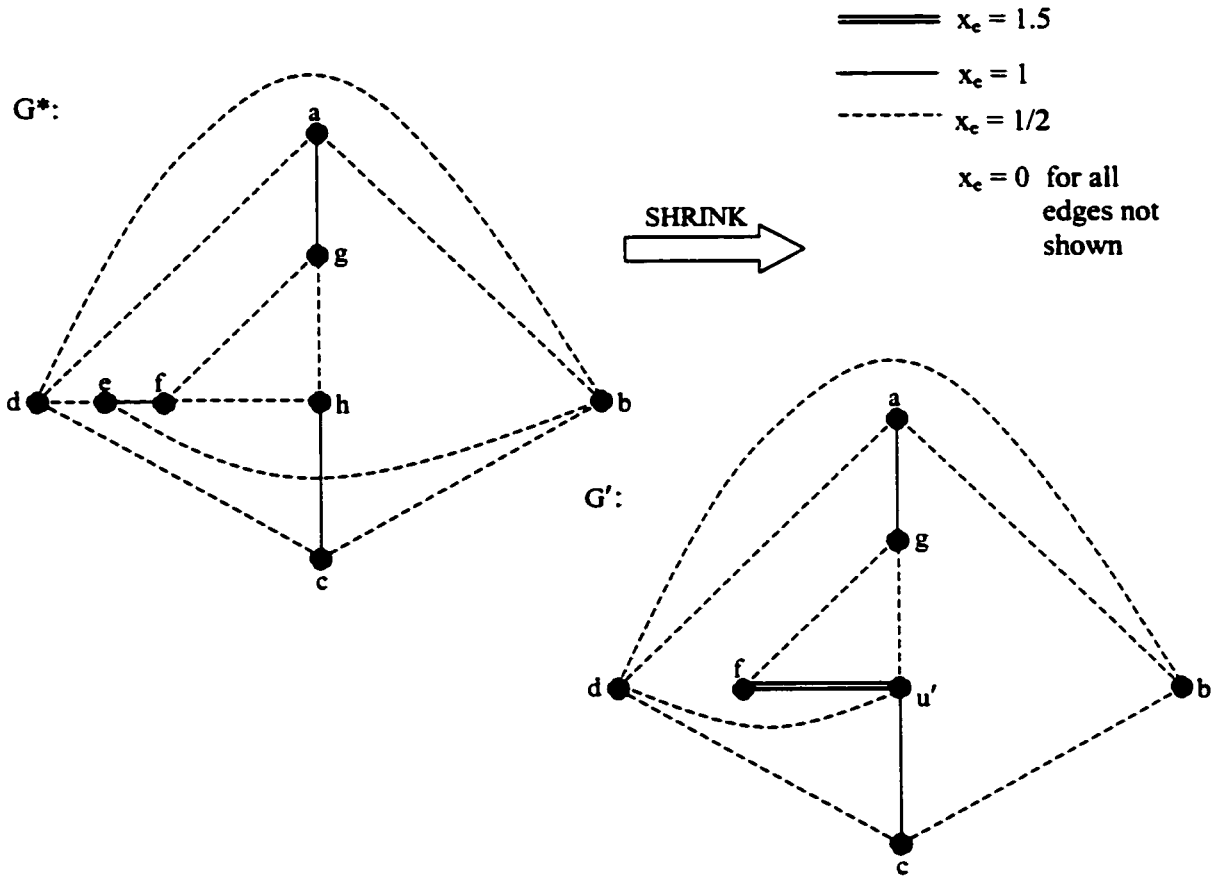


Figure 4.5: Another Shrinking Example

Having shown that for a DP-constraint in G' there is an analogous DP-constraint in G*, we must still address the details of how the analogous constraint in G* is generated in our implementation. Let the set of m shrink vertices in G' be $\{u'_1, \ldots, u'_m\}$, let $\{U_1, \ldots, U_m\}$ be

the set of disjoint subsets of vertices in G* associated with the respective shrink vertices and let $\boldsymbol{U} = \bigcup_{i=1}^{m} U_i$. We refer to the vertex sets in $\boldsymbol{U}$ as *shrink groups*.

Suppose that for a shrunk graph G', we enter Step (4h) with the DP-constraint:

$$\sum_{\substack{D'j \\ ORDINARY}} x'(\delta(D'_j)) + \sum_{\substack{D'j \\ NOT \\ ORDINARY}} x'(\delta(A'_j)) + \sum_{\substack{D'j \\ NOT \\ ORDINARY}} x'(\delta(B'_j)) + x'(\delta(H')) -$$

$$\left(\sum_{\substack{D'j \\ NOT \\ ORDINARY}} x'(E'(A'_j:B'_j)) + x'(9') - x'(F'_2)\right) \geq 3p + 1 \qquad (4.1)$$

We would like to adjust (4.1) to give the related DP-constraint for G*.

Proposition 4.2. To get the equivalent DP-constraint in G*, Step (4h) is as follows:

(a) For the vertex sets listed in (4.1) up to and including the handle, they can be written to the cuts file as usual, except that the m shrink vertices must be replaced with the corresponding shrink groups in $\boldsymbol{U}$.

(b) Fill the matrix Y, defined in Step (4h), with :

$$Y[w, z] = \begin{cases} 0, & \text{if } w, z \in U_i,\ i = 1, \ldots, m \\ Y'[w, z], & \text{if } w, z \notin \boldsymbol{U} \\ Y'[w, u'_i], & \text{if } w \notin \boldsymbol{U},\ z \in U_i,\ i = 1, \ldots, m \\ Y'[u'_i, z], & \text{if } z \notin \boldsymbol{U},\ i = 1, \ldots, m,\ w \in U_i, \\ Y'[u'_i, u'_j], & \text{if } w \in U_i,\ z \in U_j,\ i \neq j,\ i = 1, \ldots, m,\ j = 1, \ldots, m, \end{cases}$$

and proceed as usual, making the adjustments described in Chapter 3 and writing out the remaining vertex sets and right-hand side of the DP-constraint to the CONCORDE cuts file.

Proof: The proof for (a) follows from our proof for Proposition 4.1. For (b), let us consider all the cases. If w and z are in the same shrink group, Y[w, z] = 0. This follows from our definitions in the proof of Proposition 4.1. If neither w nor z are in U, Y'[w, z] is unaffected by any of the substitutions. So, Y[w, z] = Y'[w, z]. If one of w and z is in some shrink group but the other is not in any, the result again follows from the proof for Proposition 4.1. The same reasoning can be applied to the last case, where w and z are in different shrink groups. □

The determination of an adequate shrinking of a non-planar graph G* to a planar graph G' (if we can find one) deserves further discussion. Again, we resort to an ad hoc method. Recall that we use our graph viewer to determine planarity. Having decided that G* is non-planar, we examine any edge crossings and try to generate G'. Ideally, the entire planarity-testing and shrinking process should be automated.

In terms of automating planarity testing, several alternatives are available. To name one, the implementation of an O(n) algorithm that tests a graph, and if the graph is planar, exhibits a *combinatorial embedding* (that is, a representation formed from a cyclic ordering of the edges incident to each vertex), is described in [MMN93]. From here, one can use the linear-time algorithm described in [S90] to generate a corresponding straight-line embedding.

The automation of the shrinking process is much more difficult. Since we are constantly moving toward a better (closer to optimal) objective value, it is important that our shrinking process disrupt as little of G^* as possible. Thus, we would like to find shrink groups $U_1, \ldots, U_m$ for G^* such that our new graph G' is planar and $|V^*| - |V'|$ is minimized. In [WAN83], the following related problem was shown to be NP-complete (for a discussion of NP-complete problems and the closely related concept of NP-hard problems, the reader is referred to [B93]):

> Given a non-planar graph $G^* = (V^*, E^*)$ and a positive number κ, is there a subset of E^* of size at most κ such that contracting the edges in the subset results in a planar graph $G' = (V', E')$?

In this context, contracting an edge ab is equivalent to shrinking the vertex set {a, b} with one exception. Shrinking {a, b} does not require that the two vertices be connected in G^*. Notwithstanding, we strongly suspect that the decision form of our shrinking problem is also NP-complete, and currently we know of no efficient methods for solving it. Given a non-planar, straight-line embedding G^*, we now describe two possible heuristic methods. Keep in mind that ideally we would like the input graph for our heuristics to have as few edge crossings as possible. This is important for two reasons. First, disrupting as little of G^* as possible reduces the chance of failing to find a DP-constraint violated by x' when one is violated by x^*. Second, the less we have to shrink, the less work it is for us. Unfortunately, finding a drawing of a graph with the fewest number of edge crossings is NP-hard [GJ83]. In [L99], which is a comprehensive survey of general planarization techniques, a heuristic

presented in [OS94] for finding the drawing with the fewest number of edge crossings is mentioned.

The first idea is to iteratively find all pairs of edge crossings in G* and for each edge crossing, shrink one of the vertex pairs involved. For example, if edge ab crosses edge cd, then an attempt could be made to shrink any one of the pairs {a, b}, {a, c}, {a, d}, {b, c}, {b, d} and {c, d}. What follows is an outline of the algorithm:

```
find edge crossing;
still_edge_crossings := true;
while (still_edge_crossings)
begin
        process edge crossing by shrinking vertex pair;
        test planarity;
        if planar
                find straight-line planar embedding;
                still_edge_crossings := false;
        else
                find edge crossing;
end;
```

In the worst case, searching for an edge crossing requires examining each pair of edges in G* and if m is the number of edges, this can be done in $O(m^2)$ time. Recall that planarity testing and generating an embedding can both be achieved in O(n) time. Assuming G* is stored as a matrix, shrinking a vertex pair takes O(n) time. In terms of the number of edge

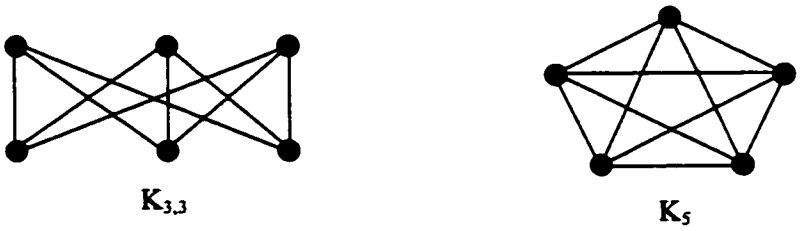
crossings, a lower bound on the number of crossings in a complete graph can be found in [L99]. We restate it here. The number of edge crossings C in a complete graph is

$$C \leq 1/4 \lfloor n/2 \rfloor \lfloor (n-1)/2 \rfloor \lfloor (n-2)/2 \rfloor \lfloor (n-3)/2 \rfloor,$$

where equality holds for $n \leq 10$. The complexity of this first heuristic is $O(Cm^2)$. Note that it is highly unlikely that the support graphs we are dealing with are complete. In practice, C was very small for our support graphs. Now the fact that there are in general very few edge crossings in our support graph implies that if not for one or two edges, our graph would be planar and thus $m \approx n$. In such cases, we can state our complexity as $O(Cn^2)$ in practice.

Before discussing the second heuristic, some background material is necessary. We define $K_{m,n}$ as the complete bipartite graph that can be partitioned into two exclusive subsets of m and n vertices such that an edge exists in $K_{m,n}$ if and only if one end vertex is in the subset of m vertices and the other is in the subset of n vertices. A *subdivision* of G* is a subgraph created by carrying out a series of *elementary operations* on G*; an elementary operation involves deleting an edge uv and inserting a vertex w along with the edges uw and vw. The following famous theorem, found in [K30], is needed.

Kuratowski's Theorem. A graph is non-planar if and only if it contains as a subgraph a subdivision of $K_{3,3}$ or K_5, shown in Figure 4.5.



$K_{3,3}$ K_5

Figure 4.6: Kuratowski graphs

The second idea requires information from the algorithm presented in [MMN93]. In addition to determining planarity, if a graph is non-planar, a Kuratowski subgraph is hi-lighted (this runs in $O(n^2)$ time). Our heuristic involves shrinking small areas of the $K_{3,3}$ or K_5 subdivision(s). For each subdivision, we start at some vertex u (of degree greater than two) and travel along the edges of the subdivision until we encounter another vertex v of degree greater than two. Then, we shrink {u,v}. The worst case complexity is again $O(Cn^2)$. The algorithm is:

```
find Kuratowski subdivison;
still_edge_crossings := true;
while (still_edge_crossings)
begin
        find vertex pair {u,v} to shrink in current Kuratowski subdivision;
        shrink {u,v};
        test planarity;
        if planar
                find straight-line planar embedding;
                still_edge_crossings := false;
        else
                find Kuratowski subdivision;
end;
```

Chapter 5

Results

To test the usefulness of DP-CUTFINDER as a tool for generating good DP-constraints for the cutting-plane method, 46 two-dimensional Euclidean TSP instances were studied using CONCORDE. One advantage to studying Euclidean problems is the fact that if x^* is the optimal solution for the subtour problem, then its support graph G^* is planar [GT91]. Thus, shrinking is not immediately required. We concentrated on two-dimensional problems because this is currently the only kind supported by our graph viewer.

The data was obtained from TSPLIB (located at http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95), a repository for several different kinds of TSP problems which come from real world problems. The naming convention for the 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 vertices in the problem. Note that the 46 problems we looked at represent all the two-dimensional Euclidean TSP instances contained in TSPLIB ranging from 51 to 1060 vertices, with the exception of two problems. Both *berlin52* and *pr107* were excluded from testing because their optimal subtour solutions were TSP tours, and therefore already solved. It should also be pointed out that the goal of this thesis was not speed or to determine how large a problem we could solve, but rather to see how effective the DP-constraints are as cutting planes for the TSP, and how effective the Letchford separation algorithm (with our enhancements) is within

cutting plane solution methods. As such, we limited our data set to graphs of no greater than about 1000 vertices. Note that testing was done on a Sun Enterprise 150 Server running Unix.

Tests T1, T2 & T3

For our testing purposes, we ran three different types of tests, which are described below.

T1 was as follows:

1) The subtour optimal was computed using CONCORDE and the file representing x* was passed to DP-CUTFINDER.
2) Using DP-CUTFINDER, DP-constraints violating x* were generated and added to a DP-constraints file.
3) The DP-constraints file was then passed to CONCORDE and re-optimized over the constraints of SEP(n) plus the DP-constraints in the file, generating a new solution x*.
4) This process was repeated until one of the three stopping criteria was met:
   - The solution vector x* was optimal for the TSP.
   - DP-CUTFINDER was unable to find any violated DP-constraints for x*.
   - The support graph for x* was non-planar and could not be shrunk using any of our heuristics.

A pictorial representation of T1 for the TSPLIB problem *lin318* can be found in Appendix D. This problem originated from a printed circuit board laser drilling problem, and was studied in [PH80].

Tests T2 and T3 were designed for comparison with the results from T1. However, unlike T1, all constraints for these two tests, not just the violated subtour constraints, were generated by CONCORDE; one iteration of CONCORDE was required per problem. In T2, options were activated in CONCORDE that restricted the kinds of constraints being added to just the combs and the subtour constraints. Here, CONCORDE is using its own routines and heuristics to find violated constraints. In T3, the CONCORDE default options were used; this set of inequalities represents those that the CONCORDE people deem most effective for solving TSP instances.

It is important to note that for all three of these tests, the final solution was forced to satisfy all the subtour constraints. The results are summarized in Table 5.1. The columns labelled T1, T2 and T3 hold the final objective values. Note that these all represent lower bounds on the optimal tour length and the closer they are to the optimal tour length (i.e. the bigger they are) the better. Several abbreviations are used: OTL is the optimal tour length; NC means that no DP-constraints were found for the last planar solution; SNC means that no DP-constraints were found for the shrunk graph of the last non-planar solution; if the last non-planar graph could not be shrunk, NP is used. The underlined values in column T1 represent the problems where DP-CUTFINDER did better (or at least as well) as CONCORDE. Notice that 37 of the 46 problems are underlined.

Problem Name	Number of Cities	Optimal Tour Length	Subtour Optimal	T1	T2	T3
eil51	51	426	422.500	OTL	OTL	OTL
st70	70	675	671.000	OTL	OTL	OTL
eil76	76	538	537.000	OTL	OTL	OTL
pr76	76	108159	105120.000	107284.333 (SNC)	107284.000	OTL

rat99	99	1211	1206.000	OTL	OTL	OTL
kroA100	100	21282	20936.500	OTL	21276.750	OTL
kroB100	100	22141	21834.000	OTL	OTL	OTL
kroC100	100	20749	20472.500	OTL	OTL	OTL
kroD100	100	21294	21141.500	OTL	OTL	21289.333
kroE100	100	22068	21799.500	OTL	OTL	22029.250
rd100	100	7910	7899.333	OTL	OTL	OTL
eil101	101	629	627.500	OTL	OTL	OTL
lin105	105	14379	14370.500	OTL	OTL	OTL
pr124	124	59030	58067.500	59009.750 (SNC)	58976.595	59014.571
bier127	127	118282	117431.000	OTL	OTL	OTL
ch130	130	6110	6075.500	OTL	OTL	OTL
pr136	136	96772	95934.500	96714.667 (NC)	96675.750	96514.667
pr144	144	58537	58189.250	OTL	OTL	OTL
ch150	150	6528	6490.125	OTL	OTL	6525.125
kroA150	150	26524	26299.000	26520.667 (NC)	26511.667	26515.417
kroB150	150	26130	25732.500	OTL	26127.500	26114.500
pr152	152	73682	73208.500	OTL	OTL	73349.500
u159	159	42080	41925.000	OTL	OTL	OTL
rat195	195	2323	2299.250	2319.586 (NP)	2318.896	2318.660
d198	198	15780	15712.000	15777.500 (NC)	15769.000	15741.167
kroA200	200	29368	29065.000	OTL	OTL	OTL
kroB200	200	29437	29165.000	OTL	OTL	29425.000
ts225	225	126643	115605.000	120756.014 (NP)	122211.354	OTL
tsp225	225	3916	3878.250	3913.667 (NC)	3913.333	3914.880
pr226	226	80369	80092.000	OTL	OTL	OTL
gil262	262	2378	2354.500	OTL (NP)	2377.000	2376.857
pr264	264	49135	49020.500	OTL	OTL	OTL
a280	280	2579	2566.000	2578.000 (NC)	2577.667	2578.000
pr299	299	48191	47380.000	48153.028 (NP)	48130.288	48186.500
lin318	318	42029	41888.750	OTL	42002.021	42003.310
rd400	400	15281	15157.000	15262.917 (SNC)	15259.391	15253.500
pr439	439	107217	105928.333	106764.100 (NP)	107007.646	107040.503
pcb442	442	50778	50499.500	50750.000 (NC)	50740.833	50750.333
d493	493	35002	34828.500	35000.667 (NC)	34978.805	34979.272

u574	574	36905	36714.000	OTL	36900.470	36894.587
rat575	575	6773	6724.000	6769.714 (NC)	6766.201	6766.310
p654	654	34643	34596.000	OTL	OTL	OTL
d657	657	48912	48455.188	48853.121 (NP)	48847.578	48872.117
u724	724	41910	41652.667	41897.167 (SNC)	41872.178	41871.685
rat783	783	8806	8772.750	OTL	8804.702	8804.932
u1060	1060	224094	222650.875	223955.145 (SNC)	223960.543	223914.463

Table 5.1: T1, T2 & T3 Results

As in [JRR95], for comparison of the lower bounds from the three tests, we used the metric

$$M = (100 * (LB - SUBT) / (OTL - SUBT)) \%,$$

where LB is the lower bound obtained from the test, SUBT is the subtour optimal value and OTL is as previously defined. Observe that if $M = 100$ %, the lower bound is optimal. The measure M reveals what percentage of the gap between the subtour optimal and TSP optimal is covered by the lower bound. On average, T1 covered 95.1 % of the gap, while T2 and T3 covered 93.6 % and 92.1 % of the gap, respectively. Thus, on average, using just the DP-constraints and the Letchford separation routine did slightly better than using the methods of CONCORDE. The results are shown in Table 5.2.

Problem Name	M values for T1	M values for T2	M values for T3
eil51	100	100	100
st70	100	100	100
eil76	100	100	100
pr76	71.2	71.2	100
rat99	100	100	100
kroA100	100	98.5	100
kroB100	100	100	100
kroC100	100	100	100
kroD100	100	100	96.9
kroE100	100	100	85.6
rd100	100	100	100
eil101	100	100	100

lin105	100	100	100
pr124	97.9	94.5	98.4
bier127	100	100	100
ch130	100	100	100
pr136	93.2	88.5	69.3
pr144	100	100	100
ch150	100	100	92.4
kroA150	98.5	94.5	96.2
kroB150	100	99.4	96.1
pr152	100	100	29.8
u159	100	100	100
rat195	85.6	82.7	81.7
d198	96.3	83.8	47.9
kroA200	100	100	100
kroB200	100	100	95.6
ts225	46.7	59.9	100
tsp225	93.8	92.9	97.0
pr226	100	100	100
gil262	100	95.7	95.1
pr264	100	100	100
a280	92.3	89.7	92.3
pr299	95.3	92.5	99.4
lin318	100	80.8	81.7
rd400	85.4	82.6	77.8
pr439	64.9	83.8	86.3
pcb442	89.9	86.7	90.1
d493	99.2	86.6	86.9
u574	100	97.6	94.5
rat575	93.3	86.1	86.3
p654	100	100	100
d657	87.1	85.9	91.3
u724	95.0	85.3	85.1
rat783	100	96.1	96.8
u1060	90.4	90.8	87.6

Table 5.2: Comparison of Lower Bounds for T1, T2 & T3

A more detailed look at the T1 results can be seen in Table 5.3, where we report which iterations of the process had solutions x* with non-planar support graphs and how many of the violated DP-constraints found were actually comb constraints. We also report on the number of *tight* (i.e. satisfied at equality) DP-constraints for our final solution x*. These represent the DP-constraints that are required and necessary for the final LP and solution x*. Several observations are key. Notice that for just under half of the problems, non-planar support graphs were encountered along the way. The importance of being able to shrink these points is underlined by the fact that doing this allowed us to continue moving toward a better (larger) lower bound. In fact, for six of the problems where non-planar graphs arose, optimality was eventually reached; these are shown in bold.

DP-CUTFINDER generates two kinds of DP-constraints: comb inequalities and other inequalities. We refer to these latter inequalities as *non-combs*. From Table 5.3, it is worth noting that for every problem, more combs were found than non-combs. On average, of the total number of DP-constraints found, 21.9 % were non-combs. Also note that of the tight DP-constraints, an average of 20.8 % were non-combs. Thus, non-comb constraints were definitely required and active for our final solutions.

Problem Name	Number of Iterations of CONCORDE	Iterations where Non-planar Solution	Total Number of DP-cuts (Combs / Non-combs)	Total Number of Tight DP-cuts (Combs / Non-combs)
eil51	7	----	(40 / 10)	(32 / 7)
st70	4	----	(15 / 3)	(9 / 2)
eil76	5	----	(19 / 1)	(8 / 0)
pr76	5	4, 5	(31 / 13)	(22 / 7)
rat99	7	----	(19 / 5)	(11 / 1)

kroA100	7	----	(23 / 9)	(22 / 9)
kroB100	9	----	(45 / 6)	(14 / 3)
kroC100	5	----	(28 / 2)	(8 / 2)
kroD100	4	----	(27 / 4)	(26 / 4)
kroE100	**10**	**5, 6, 7, 8**	**(87 / 29)**	**(69 / 22)**
rd100	3	----	(17 / 4)	(15 / 4)
eil101	3	----	(17 / 5)	(15 / 4)
lin105	2	----	(1 / 0)	(1 / -)
pr124	11	3, 4, 5, 9, 10, 11	(12 / 0)	(12 / -)
bier127	**7**	**3, 6**	**(51 / 11)**	**(42 / 8)**
ch130	9	----	(60 / 11)	(52 / 11)
pr136	10	3, 6, 7	(130 / 30)	(94 / 8)
pr144	3	----	(18 / 10)	(13 / 10)
ch150	7	----	(91 / 15)	(74 / 9)
kroA150	10	----	(98 / 31)	(65 / 17)
kroB150	8	----	(90 / 17)	(84 / 14)
pr152	5	----	(37 / 8)	(31 / 6)
u159	5	----	(14 / 0)	(9 / -)
rat195	11	3, 5, 10, 11	(196 / 69)	(44 / 13)
d198	11	----	(158 / 41)	(82 / 20)
kroA200	10	----	(163 / 88)	(133 / 83)
kroB200	7	----	(83 / 12)	(69 / 8)
ts225	6	3, 4, 5, 6	(51 / 4)	(10 / 0)
tsp225	13	----	(261 / 94)	(221 / 83)
pr226	4	----	(14 / 2)	(14 / 2)
gil262	**15**	**3, 5, 6, 7, 8, 9, 11, 13, 14, 15**	**(310 / 201)**	**(34 / 16)**
pr264	5	----	(23 / 3)	(23 / 3)
a280	10	----	(123 / 31)	(71 / 23)
pr299	14	5, 6, 7, 8, 9, 10, 11, 12, 13, 14	(229 / 126)	(44 / 16)
lin318	**8**	**3, 4, 5, 6**	**(157 / 113)**	**(104 / 83)**
rd400	16	4, 5, 6, 7, 9, 11, 12, 13, 14, 15, 16	(425 / 285)	(247 / 208)
pr439	6	2, 3, 4, 6	(131 / 35)	(16 / 2)
pcb442	16	4, 5, 6	(265 / 40)	(133 / 16)
d493	45	34, 40	(1505 / 800)	(408 / 120)
u574	**12**	**9, 11**	**(507 / 393)**	**(464 / 365)**

rat575	16	----	(711 / 406)	(495 / 259)
p654	3	----	(18 / 8)	(6 / 0)
d657	12	5, 6, 8, 9, 10, 11, 12	(685 / 327)	(364 / 172)
u724	16	5, 7, 8, 9, 10, 11, 13, 14, 15, 16	(892 / 591)	(602 / 356)
rat783	**13**	**11, 12**	**(752 / 249)**	**(490 / 102)**
u1060	16	8, 9, 15, 16	(674 / 218)	(144 / 10)

Table 5.3: A More Detailed Look at T1 Results

Test T4

We ran one more test, called T4, on our data. The purpose of this test was to see if DP-CUTFINDER could "cut off" any of the final solutions found by CONCORDE alone. For T4, we used the x-vector that was produced by T3 (the test using the CONCORDE default options) as our starting solution. DP-CUTFINDER was run on this point once, the violated DP-constraints were placed in a file, and then CONCORDE was run once as well to re-optimize with these DP-constraints added. Note that the only active inequalities used by CONCORDE were the DP-constraints and the subtour constraints. Again, CONCORDE was set up such that no subtour constraints were violated by the new solution.

The results are shown in Table 5.4. A single asterisk denotes a problem where T3 generated a non-planar support graph (that had to be shrunk before carrying out T4). As for a double asterisk, it signifies that T4 did not yield an increase in the lower bound, although x^* changed (this occurred for two of the problems). We attempted to rectify this situation by removing the condition that the final solution had to satisfy all the subtour constraints. In both problems, this had no effect on the lower bound.

Note from the results in Table 5.4 that for all the problems except *tsp225* and *a280*, DP-CUTFINDER found DP-constraints which cut off the final x* from CONCORDE, and in all of these cases some of the violated DP-constraints found were for combs. On average, the increase in the lower bound gap percentage M (described earlier) achieved by this one iteration of DP-CUTFINDER and re-optimizing was 3.5 %.

Another interesting observation from Table 5.4 deals specifically with the problems *kroD100*, *kroB200* and *pr439*. For all three points T4 yielded an increase in the lower bound, yet none of the violated DP-constraints we found for these problems were tight for the final solution. At first this seemed quite puzzling, so we checked our DP-constraints by hand, and verified that they were indeed violations of the original x-vectors and that they were not tight for the final x-vectors. We suspect that these anomalies were caused by the preprocessing of LPs by CONCORDE, as we know that CONCORDE sometimes ignores some of the cuts in a cut file (when solving the LP) if they do not appear to be useful on a sparser version of the graph.

Problem Name	Optimal Tour Length	Initial Objective Value	Final Objective Value	Number of DP-cuts (Combs / Non-combs)	Total Number of Tight DP-cuts (Combs / Non-combs)
kroD100	21294	21289.333	21290.800	(2 / 0)	(0 / -)
kroE100 *	22068	22029.250	22038.909	(9 / 0)	(6 / -)
pr124 *	59030	59014.571	59025.639	(12 / 0)	(11 / -)
pr136	96772	96514.667	96603.600	(10 / 0)	(10 / -)
ch150	6528	6525.125	6527.067 (OTL)	(9 / 9)	(2 / 0)
kroA150	26524	26515.417	26517.667	(13 / 2)	(12 / 2)
kroB150	26130	26114.500	OTL	(19 / 13)	(19 / 13)

pr152	73682	73349.500	73362.300	(4 / 0)	(2 / -)
rat195	2323	2318.660	2319.454	(3 / 0)	(2 / -)
d198	15780	15741.167	15756.250	(15 / 2)	(12 / 0)
kroB200	29437	29425.000	29431.708	(2 / 0)	(0 / -)
tsp225	3916	3914.880	----	NC	(- / -)
gil262 *	2378	2376.857	2377.000	(4 / 0)	(4 / 0)
a280	2579	2578.000	----	NC	(- / -)
pr299	48191	48186.500	48188.000	(1 / 0)	(1 / -)
lin318 *	42029	42003.310	42021.300	(18 / 40)	(17 / 19)
rd400	15281	15253.500	15255.067	(7 / 0)	(1 / -)
pr439 *	107217	107040.503	107055.051	(5 / 2)	(0 /0)
pcb442 **	50778	50750.333	50750.333	(4 / 0)	(4 / -)
d493	35002	34979.272	34985.525	(50 / 42)	(15 / 3)
u574	36905	36894.587	36897.629	(30 / 9)	(12 / 1)
rat575	6773	6766.310	6767.671	(28 / 5)	(4 / 1)
d657 *	48912	48872.117	48877.603	(30 /56)	(3 / 0)
u724 *	41910	41871.685	41873.843	(15 / 1)	(2 / 0)
rat783 **	8806	8804.932	8804.932	(3 / 0)	(1 / -)
u1060	224094	223914.463	223921.907	(24 / 7)	(15 / 2)

Table 5.4: T4 Results

Tests T5, T6, T7 & T8: *lin318*

The problem *lin318* was the subject of the last four tests. Tests T5 and T6 were slight variations of T1. In T5, only violated comb constraints found by DP-CUTFINDER were added to the constraints file. The motivation behind this test was to determine if restricting the kinds of DP-constraints being added to comb constraints would help us in maintaining a planar graph. For this particular example, it was not the case.

In T6, only the most violated inequalities were permitted in the constraints file, just as Letchford had originally set out in [LE00]. The purpose of this test was to see if our enhancement of using all the violations found per iteration of DP-CUTFINDER, as opposed

to just the maximum violation(s), was worthwhile. For this particular problem, 26 iterations of CONCORDE were required for T6, versus just 8 iterations for T1.

In both T5 and T6, optimality for the TSP was achieved. The results are shown in Table 5.5.

lin318	Number of Iterations of CONCORDE	Iterations where Non-planar Solution	Total Number of DP-cuts (Combs / Non-combs)	Total Number of Tight DP-cuts (Combs / Non-combs)
T5	9	2, 3, 4, 5	(175 /-)	(139 / -)
T6	26	9, 10, 11, 14, 19, 22, 24, 25	(128 / 43)	(102 / 38)

Table 5.5: T5 & T6 Results

Test T7 was similar to T3, but the code ensuring that the final solution satisfied all the subtour constraints was removed. It had been suggested to us by the makers of CONCORDE that CONCORDE's results were often better if the condition that the subtour constraints had to be satisfied was relaxed. For *lin318*, this was indeed true; test T7 yielded a lower bound of 42014.429, while T3 resulted in a lower bound of 42003.310.

Test T8 was an extension of T4. We started with the x-vector found by CONCORDE alone in T3, and then repeatedly ran DP-CUTFINDER and re-optimized using CONCORDE until one of the stopping criteria was reached. TSP optimality was reached after 3 iterations (see Table 5.6).

lin318	Number of Iterations of CONCORDE	Iterations where Non-planar Solution	Total Number of DP-cuts (Combs / Non-combs)	Total Number of Tight DP-cuts (Combs / Non-combs)
T8	3	1	(56 / 45)	(55 / 24)

Table 5.6: T8 Result

Chapter 6

Conclusion

In this thesis, we presented the enhancements we made to Adam Letchford's exact separation algorithm for DP-constraints. Most significantly, we described a method for preventing the algorithm from halting in the case where we have a non-planar support graph. We also improved the algorithm efficiency (in terms of number of iterations) by returning more violated DP-constraints per iteration than the original. In addition to these enhancements, we showed how the cycles found in the separation routine could be translated into a closed set form, and in particular into a form which is "user-friendly" for the software package CONCORDE, which we used. As well, we tested our implementation, DP-CUTFINDER, on a selected set of 46 problems from TSPLIB (test T1) and compared our results to those obtained by CONCORDE using both comb-type cuts (test T2) and the default cuts for CONCORDE (test T3). Below we summarize some of our results and the corresponding conclusions.

Result 1: For 37 of the 46 problems studied, our lower bounds obtained in test T1 were as good as or better than the ones found by CONCORDE in tests T2 and T3. Also, on average, the values from test T1 covered slightly more of the gap between the subtour optimal value and the optimal tour length than the values of tests T2 and T3.

Conclusion: Using only DP-constraints, our enhanced version of Letchford's algorithm performed very well in terms of solution quality when compared to the solutions produced by a state-of-the-art tool such as CONCORDE.

Result 2: Many of the DP-constraints found and used in test T1 were tight, and thus required for the final solutions obtained. Of these tight DP-constraints, roughly 21% were non-combs.

Conclusion: Non-comb DP-constraints were definitely required and active for our final solutions.

Result 3: For just under half of the 46 problems, non-planar support graphs were encountered, often very early in the cutting-plane process. In these cases, we were able to use our shrinking algorithm to continue moving towards a better lower bound.

Conclusion: Being able to shrink and use the algorithm for non-planar support graphs is a very useful and important enhancement for the Letchford algorithm.

Result 4: In test T4, we checked to see if DP-CUTFINDER could cut off any of the 26 final solutions found by CONCORDE (using their default options) for the problems for which CONCORDE did not find the optimal tour. Of these 26 problems, we were able to improve the lower bound for all but two of them. Both violated combs and non-combs were found by DP-CUTFINDER.

Conclusion: Once again, this shows that DP-constraints are useful as cuts. It also shows that the enhanced Letchford algorithm was able to find violated combs missed by the heuristic comb separation algorithms used by CONCORDE.

In summarizing the above conclusions, we feel our results have demonstrated the practical usefulness of the DP-constraints and the enhanced Letchford algorithm within a branch and cut framework for the TSP.

We have several ideas for topics of future research, the most important of which would be to automate the planarity testing and shrinking processes and incorporate them into DP-CUTFINDER. The development and implementation of other shrinking heuristics for the more complicated non-planar support graphs would also be beneficial.

In terms of testing, our results showed that the use of DP-constraints as effective cutting planes for the TSP has potential. However, to further substantiate our claim, the separation algorithm should be applied to a larger, more complete set of problems. One of the difficulties we encountered with our implementation was dealing with problems with more than 1000 vertices, due to the fact that our implementation of the Letchford algorithm is quite slow. As mentioned in Chapter 4, there are many ways in which the efficiency of our implementation could be improved if more effort was put into programming (for example, more efficient algorithms exist for the steps for finding the edges in the planar dual, and for finding the shortest s to t paths). We believe that there are many ways in which the Letchford algorithm could be implemented more efficiently than it was in DP-

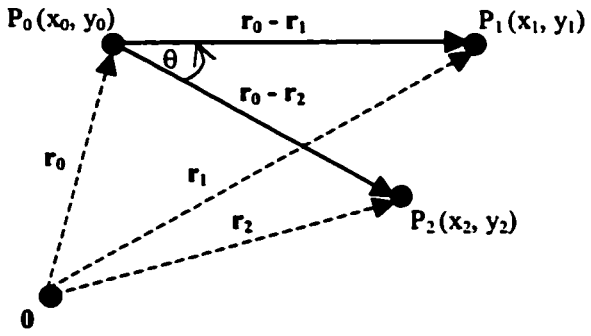
CUTFINDER, however the programming effort would be substantial. Also, in [NT98], some simple techniques for shrinking graphs are discussed. They are all based on shrinking sets of vertices along paths of edges (with each edge having cost 1) into shorter paths of vertices. Applying these methods to some of our problem instances, both planar and non-planar, would reduce the problem size and thus enable us to test DP-CUTFINDER on some larger examples.

Finally, it would be interesting to carry out test T8 on some different problems. Recall that this was the test where we tried to improve on the x-vector resulting from applying the CONCORDE default options to a problem until one of the stopping criteria was met. Good results here would be significant in that this would solidify our findings that the DP-constraints form a useful class of cutting planes for the TSP.

Appendix A

Angle Calculations for Dual-finding Algorithm

To measure the angle θ, shown below, in the counterclockwise direction with respect to P_0P_2, we use the formulas for the cosine and sine functions, which can be found in [CO74].



$$\begin{aligned}\cos\theta &= ((\mathbf{r}_0 - \mathbf{r}_1) \bullet (\mathbf{r}_0 - \mathbf{r}_2)) / (|\mathbf{r}_0 - \mathbf{r}_1|\ |\mathbf{r}_0 - \mathbf{r}_2|) \\ &= (\mathbf{r}_0^2 - \mathbf{r}_0 \bullet \mathbf{r}_2 - \mathbf{r}_1 \bullet \mathbf{r}_0 + \mathbf{r}_1 \bullet \mathbf{r}_2) / (|\mathbf{r}_0 - \mathbf{r}_1|\ |\mathbf{r}_0 - \mathbf{r}_2|) \\ &= ((x_0^2 + y_0^2) - (x_0x_2 + y_0y_2) - (x_1x_0 + y_1y_0) + (x_1x_2 + y_1y_2)) / \\ &\qquad \sqrt{((x_0 - x_1)^2 + (y_0 - y_1)^2)}\ \sqrt{((x_0 - x_2)^2 + (y_0 - y_2)^2)}\end{aligned}$$

$$\begin{aligned}\sin\theta &= |(\mathbf{r}_0 - \mathbf{r}_1) \times (\mathbf{r}_0 - \mathbf{r}_2)| / (|\mathbf{r}_0 - \mathbf{r}_1|\ |\mathbf{r}_0 - \mathbf{r}_2|) \\ &= ((\mathbf{r}_1 \times \mathbf{r}_2) - (\mathbf{r}_1 \times \mathbf{r}_0) - (\mathbf{r}_0 \times \mathbf{r}_2)) / (|\mathbf{r}_0 - \mathbf{r}_1|\ |\mathbf{r}_0 - \mathbf{r}_2|) \\ &= ((x_1y_2 - x_2y_1) - (x_1y_0 - x_0y_1) - (x_0y_2 - x_2y_0)) / \\ &\qquad \sqrt{((x_0 - x_1)^2 + (y_0 - y_1)^2)}\ \sqrt{((x_0 - x_2)^2 + (y_0 - y_2)^2)}\end{aligned}$$

There are two possibilities for the angle measurement using acos(θ). Either, θ = acos(θ) or θ = 360° – acos(θ). Using asin(θ), there are also two possibilities. Either, θ = asin(θ) or θ = 180° – asin(θ). The final angle measurement is the matching value between the two different sets of possibilities.

Appendix B

Theorems for Planar Graphs

We restate *Euler's Formula*, as found in [R91]:

If G is a simple, connected, planar graph with n vertices, m edges and f faces, then

$$n - m + f = 2 .$$

Another theorem, which can also be found in [R91], is:

If G is a simple, connected, planar graph with vertex set V and m edges, then for all $u \in V$

$$\sum \ degree(u) = 2m . \qquad \text{(B.1)}$$

(i.e. when the degrees are summed, each edge adds one to the counts of each of the two vertices on its endpoints)

We use the above theorems to prove the following proposition.

Proposition. If n and m are as defined above, then $m \in \Theta(n)$.

Proof: Define the planar support graph $G^* = (V^*, E^*)$ and its planar dual $\underline{G}^* = (\underline{F}^*, \underline{R}^*)$ as usual. Then,

$$|\underline{R}^*| = |E^*| . \qquad \text{(B.2)}$$

We define the degree of a face as the number of edges in that face.

From (B.1) and (B.2),

$$\sum_{v \in \underline{F}^*} \text{degree}(v) = 2\,|\underline{R}^*| = 2\,|E^*| . \qquad \text{(B.3)}$$

Let the degree of each face be greater or equal to k. From (B.3),

$$\sum_{v \in \underline{F}^*} \text{degree}(v) = 2\,|E^*| \geq k\,|\underline{F}^*|\,.$$

Assuming that k is at least 3 (assuming no multiple edges or loops),

$$|\underline{F}^*| \leq 2/3\,|E^*|\,. \qquad \text{(B.4)}$$

Using Euler's Formula,

$$|V^*| + |\underline{F}^*| = |E^*| + 2\,. \qquad \text{(B.5)}$$

Then, substituting (B.4) into (B.5), we have the following derivation:

$$|V^*| + 2/3\,|E^*| \geq |E^*| + 2$$

$$|V^*| - 2 \geq 1/3\,|E^*|$$

$$|E^*| \leq 3\,|V^*| - 6\,.$$

Since we also know that $|E^*| \geq |V^*| - 1$ for any connected graph, it follows that $m \in \Theta(n)$.

Proposition. If n and f are as previously defined, $f \in O(n)$.

Proof: Inequality (B.4) can be written as:

$$|E^*| \geq 3/2\,|\underline{F}^*|\,.$$

Substituting into (B.5), we have the derivation:

$$|V^*| + |\underline{F}^*| \geq 3/2\,|\underline{F}^*| + 2$$

$$|V^*| \geq 1/2\,|\underline{F}^*| + 2$$

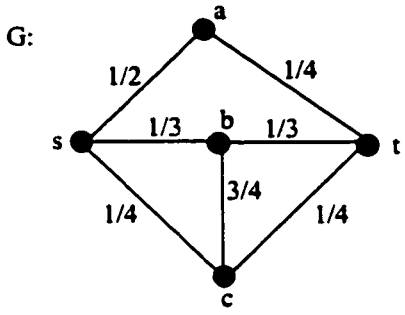
$$|\underline{F}^*| \leq 2\,|V^*| - 4\,.$$

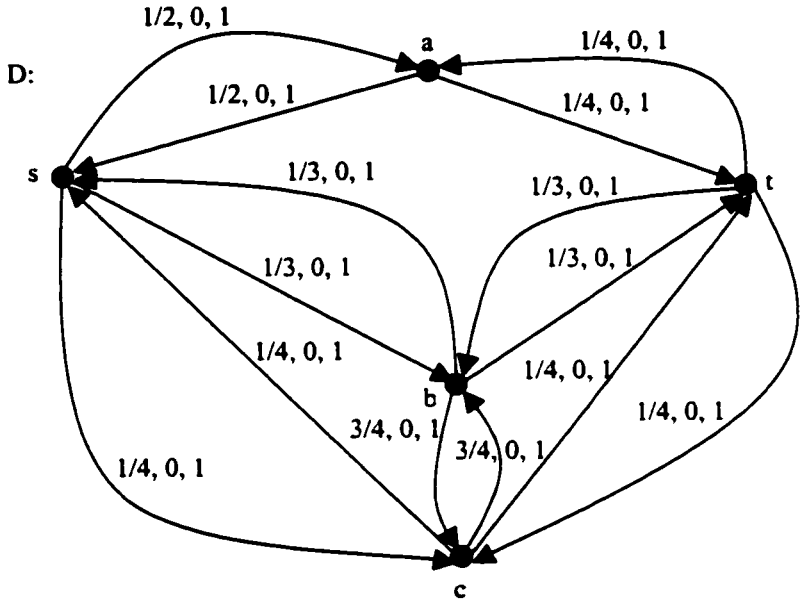
Therefore, $f \in O(n)$. Note that since we only have an upper bound on f, we cannot say anything stronger.

Appendix C

Minimum Cost Flow Example

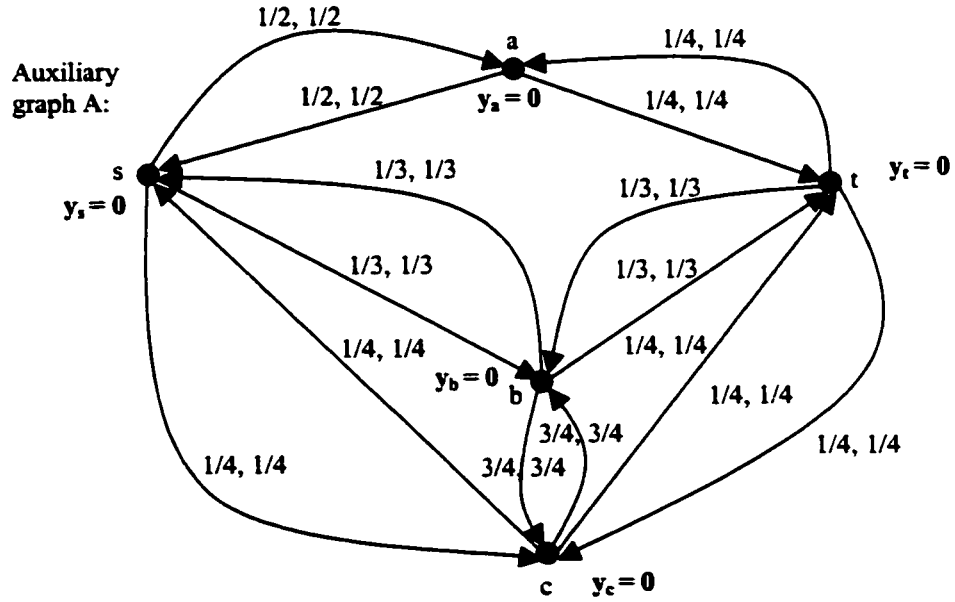
To find a minimum cost s to t to flow (of value 3) in a graph G, start by building a digraph D from G as shown below (the 3 labels on each edge in D represent the cost, flow and capacity respectively):





Iteration 1:

Initialize FV = 0, w_{st} = 0, y = **0**, X = **0**.



Labelling of arcs in A : c_{ij}, c^y_{ij} , where c_{ij} is calculated using procedure check_arc_ij and

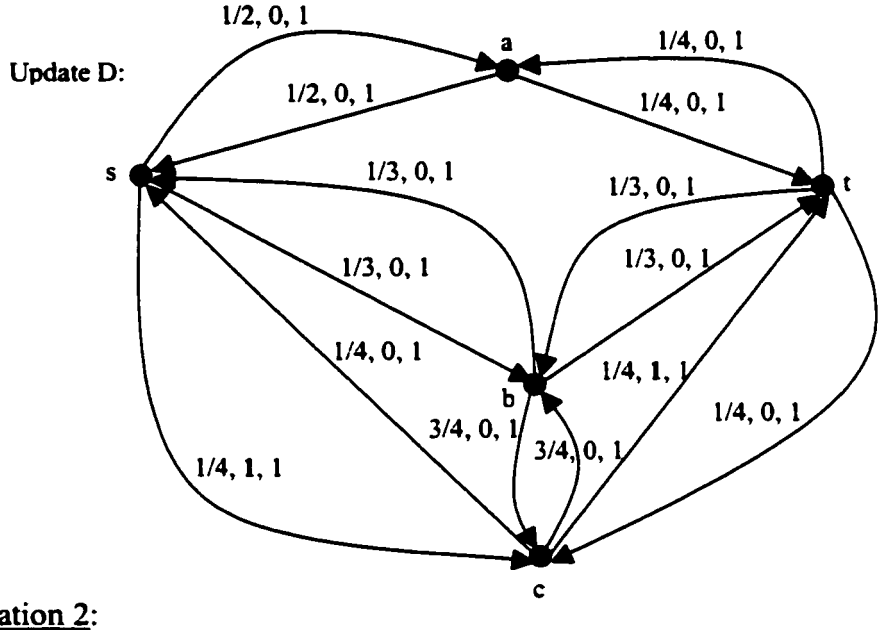
$c^y_{ij} = c_{ij} - y_i + y_j$.

Use c^y_{ij} as the costs for shortest path algorithm, but note that on the first iteration: $c_{ij} = c^y_{ij}$.

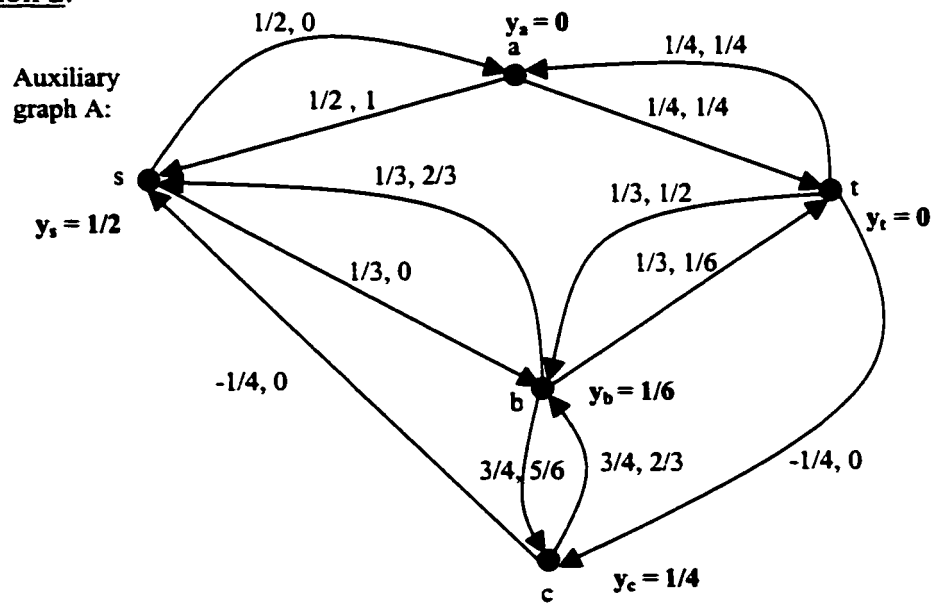
Apply Dijkstra's Shortest Path Algorithm to obtain the shortest s to t path P = s, c, t; vertices permanently labelled: s, c, b, t.

$d_s = 0$; $d_b = 1/3$; $d_c = ¼$; $d_t = ½$, $\varepsilon = 1$, X[s, c] = 1; X[c, t] = 1; FV = 1; $w_{st} = ½$;

Adjust potentials: $y_s = ½$; $y_b = 1/6$; $y_c = ¼$; $y_t = 0$;



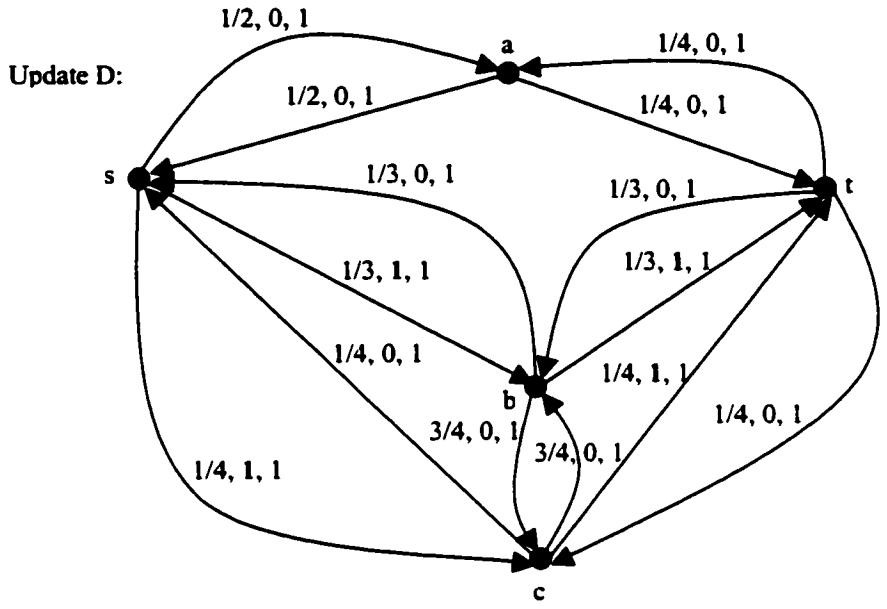
Iteration 2:



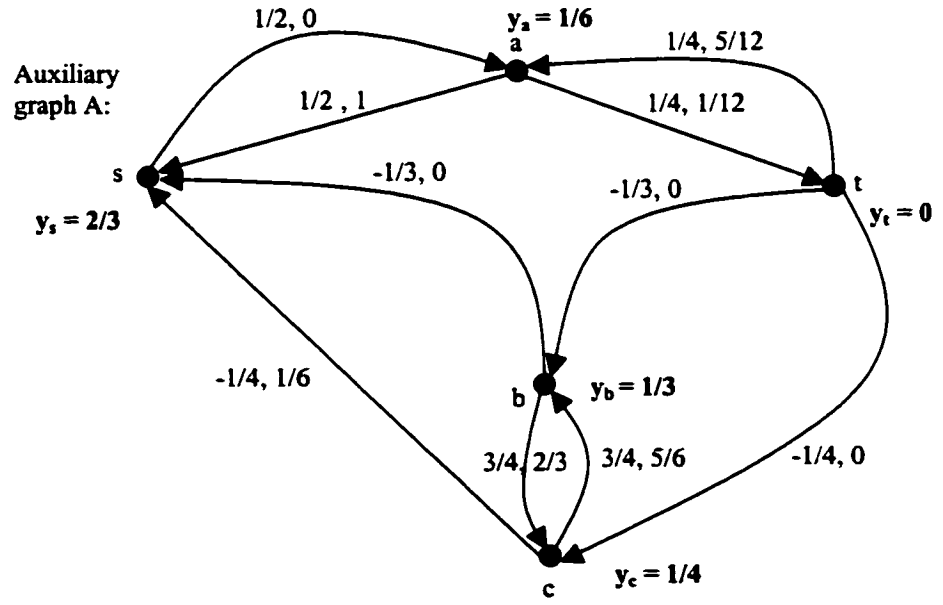
P = s, b, t; vertices permanently labelled in Dijkstra's Shortest Path Algorithm: s, a, b, t.

$d_s = 0$; $d_a = 0$; $d_b = 0$; $d_t = 1/6$, $\varepsilon = 1$, X[s, b] = 1; X[b, t] = 1; FV = 2; $w_{st} = ½ + (1/6+½) = 7/6$;

Adjust potentials: $y_s = 2/3$; $y_a = 1/6$; $y_b = 1/3$; $y_t = 0$;



Iteration 3:



P = s, a, t; vertices permanently labelled in Dijkstra's Shortest Path Algorithm: s, a, t.

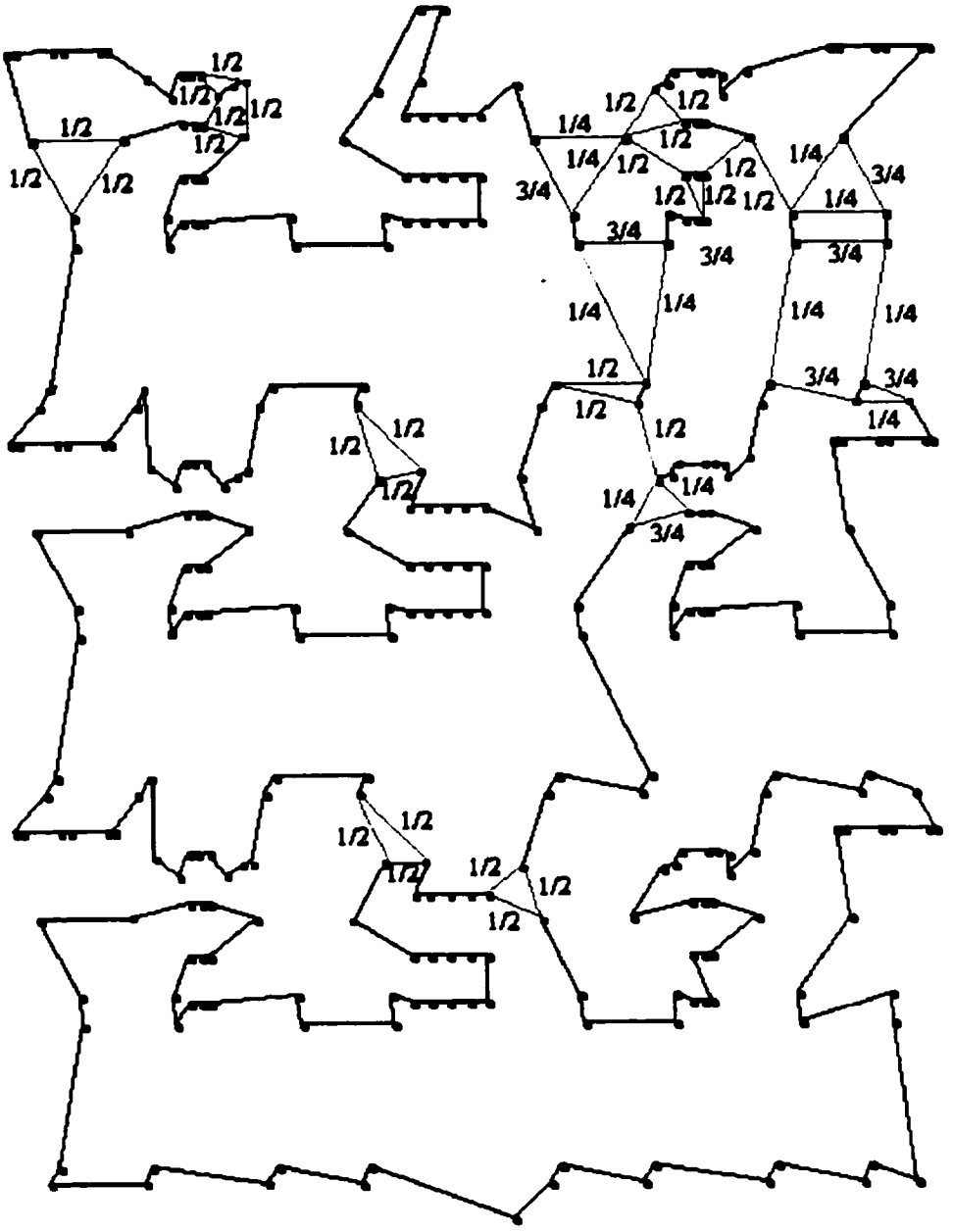
$d_s = 0$; $d_a = 0$; $d_t = 1/12$, $\varepsilon = 1$, X[s, a] = 1; X[a, t] = 1; FV = 3; $w_{st} = 7/6+(1/12+2/3) = 23/12$;

Since FV = 3, stop. The three edge-disjoint paths of minimum total weight $w_{st} = 23/12$ are

s, a, t, and s, b, t and s, c, t.

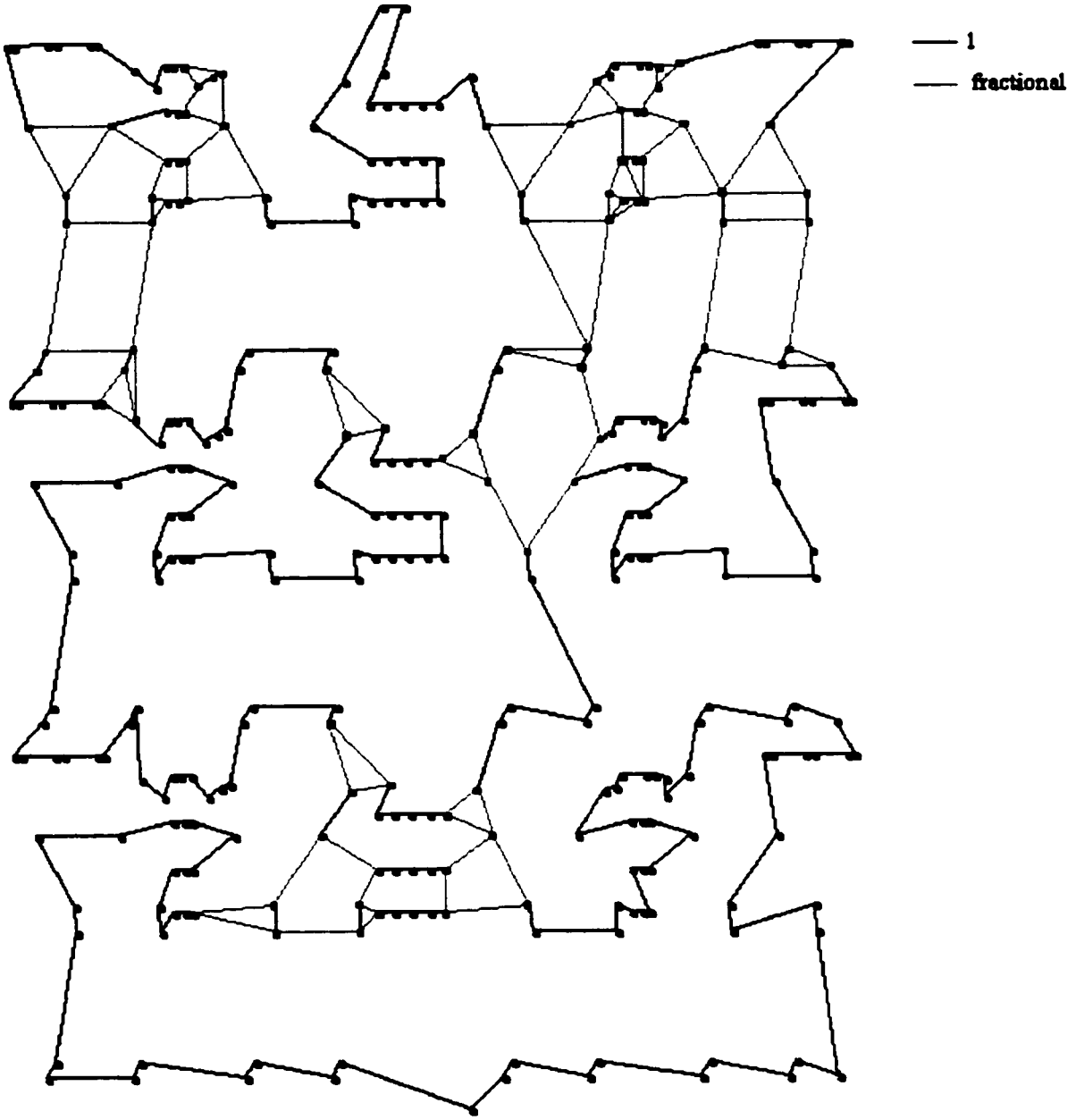
Appendix D

Test T1 FOR lin318



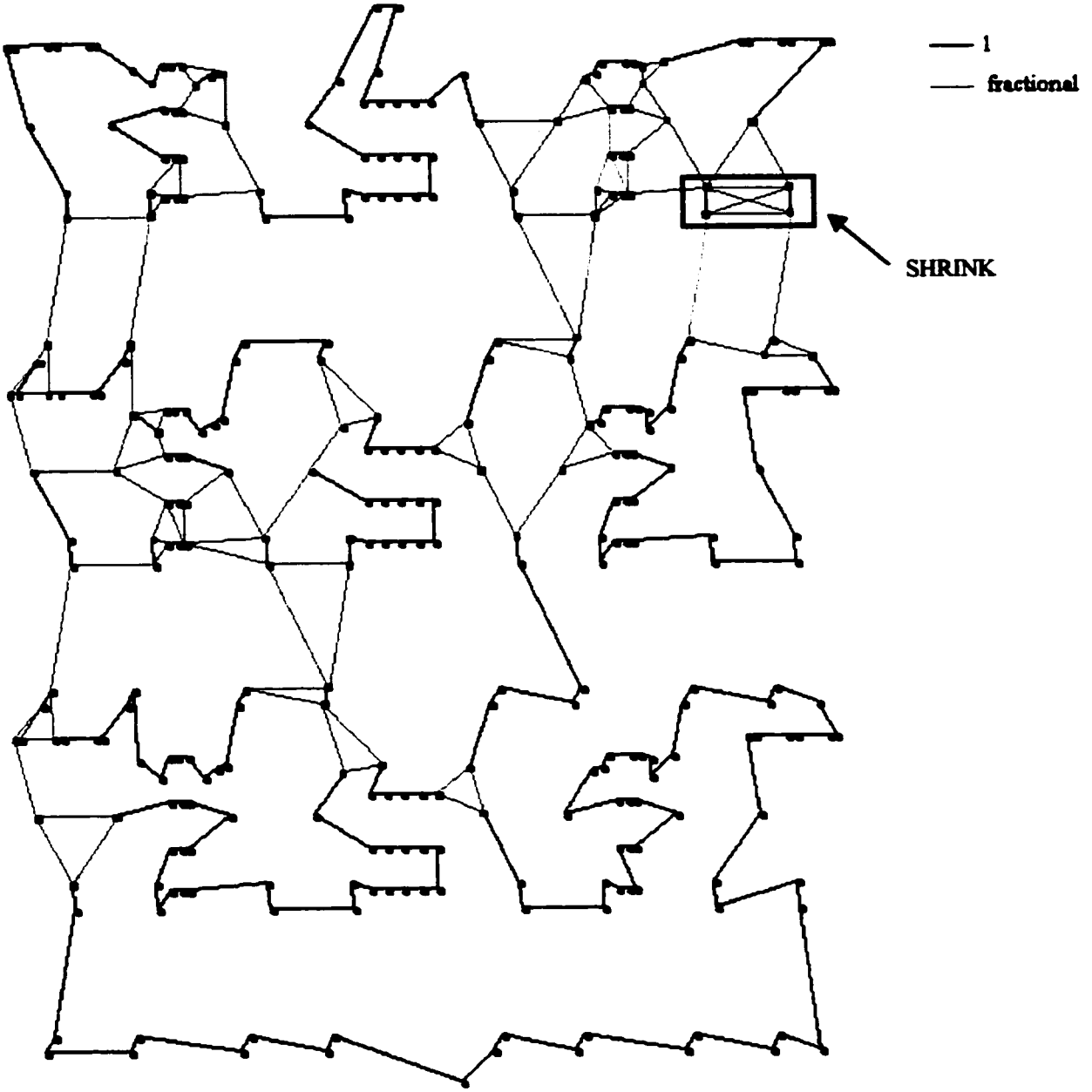
CONCORDE ITERATION 1
SUBTOUR OPTIMAL OBJECTIVE VALUE: 41 888.750

AFTER ADDING SOME DP-CONSTRAINTS ...



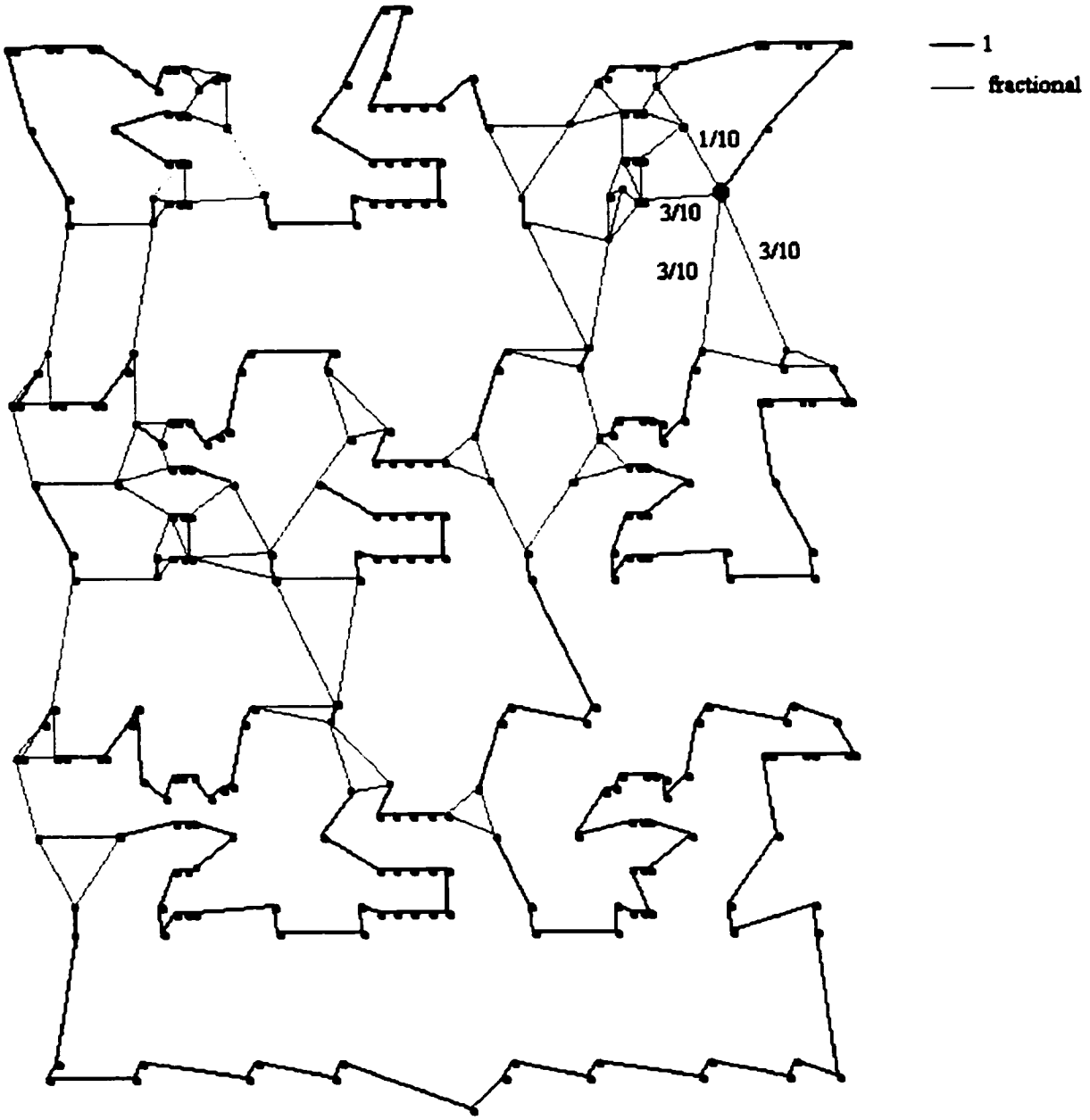
CONCORDE ITERATION 2
OBJECTIVE VALUE: 41 918.000

AFTER ADDING MORE DP-CONSTRAINTS ...



A NON-PLANAR SOLUTION

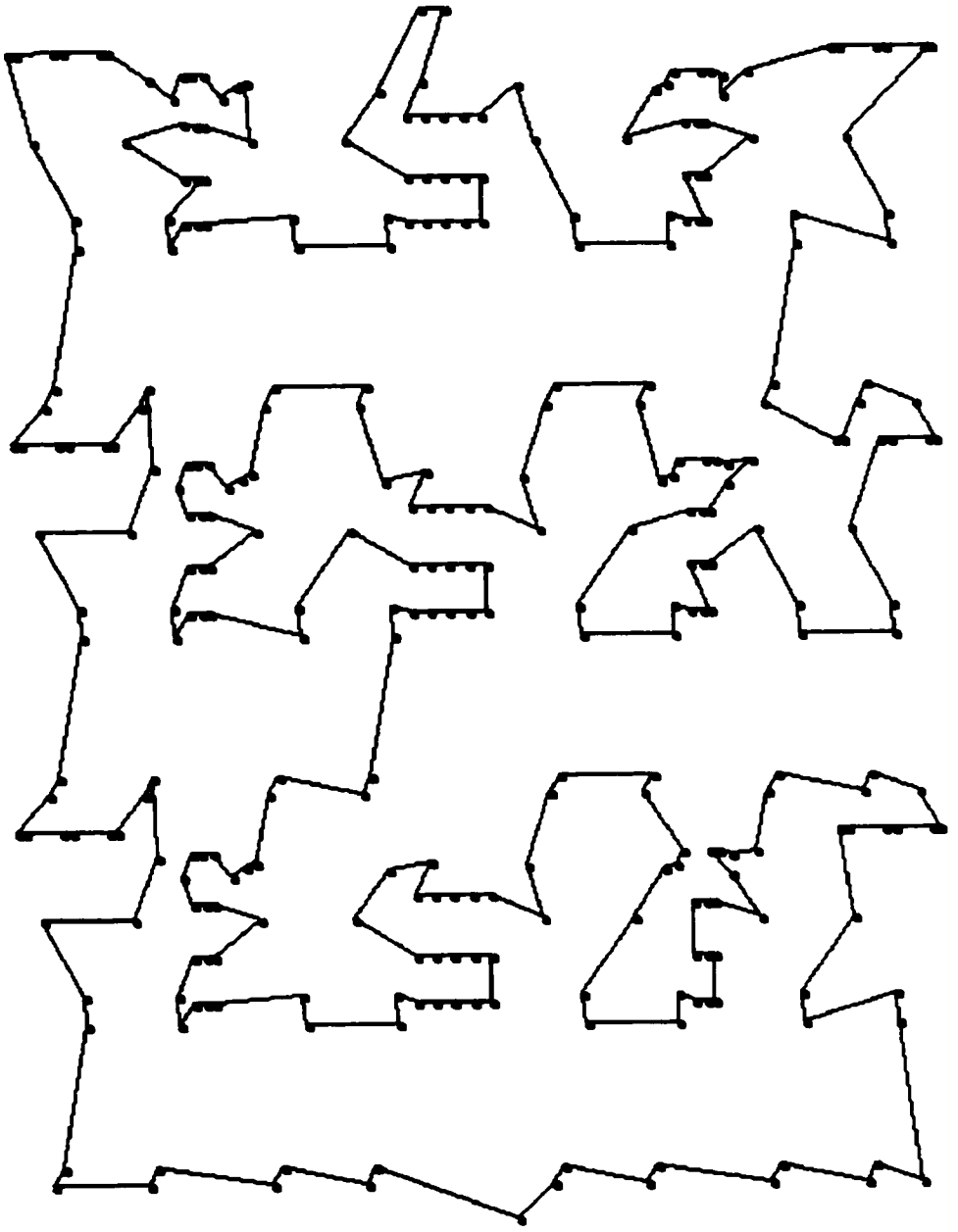
AFTER SHRINKING POINT ...



CONCORDE ITERATION 3
OBJECTIVE VALUE: 41 928.15

Note that in this particular case we did not use either of our heuristics to shrink, but rather shrunk the entire set of four nodes involved in the edge crossing.

AFTER 8 ITERATIONS OF DP-CUTFINDER AND CONCORDE ...



AN OPTIMAL TSP SOLUTION
OBJECTIVE VALUE: 42 029. 000

References

[ABCC95] D. Applegate, R. Bixby, V. Chvátal and W. Cook, "Finding Cuts in the TSP (A preliminary report)", DIMACS 95-05, 1995, 64 pages.

[AL00] A. Letchford (2000), Private communication.

[AMO93] R.K. Ahuja, T.L. Magnanti and J.B. Orlin, "Network Flows: Theory, Algorithms, and Applications ", Prentice-Hall Inc., 1993.

[B93] S. Baase, "Computer Algorithms: Introduction to Design and Analysis", Addison-Wesley Publishing Company, 1993.

[BC91] S. Boyd and W. Cunningham, "Small traveling salesman polytopes", *Mathematics of Operations Research* 16, 1991, pg. 259-271.

[BCCN96] E. Balas, S. Ceria, G. Cornuéjols and N. Natraj, "Gomory cuts revisited", *Operations Research Letters* 19, 1996, pg. 1-9.

[C97] R. Carr, "Separating clique trees and bipartition inequalities having a fixed number of handles and teeth in polynomial time", *Mathematics of Operations Research* 22-2, 1997, pg. 257-265.

[C01] S. Cockburn, "On DP-Constraints for the Traveling Salesman Polytope", Master's Thesis, University of Ottawa, Ottawa, 2001.

[CFL00] A. Caprara, M. Fischetti and A. Letchford, "On the separation of maximally violated mod-k cuts", *Math. Program.* 87, 2000, pg. 37-56.

[CO74] R. Cole, "Vector Methods", Van Nostrand Reinhold Company Limited, London, 1974.

[DFJ54] G. Dantzig, R. Fulkerson and S. Johnson, "Solution of large-scale traveling-salesman problem", *Operations Research* 2, 1954, pg. 393-410.

[E65] J. Edmonds, "Maximum matching and a polyhedron with 0,1 vertices", *Journal of Research of the National Bureau of Standards* (B) 69, 1965, pg. 125-130.

[FT99] L. Fleischer and I. Tardos, "Separating maximally violated comb inequalities", *Math. Oper. Res* 24, 1999, pg. 130-148.

[GJ83] M. Garey and D. Johnson, "Crossing number is NP-complete", *SIAM J. Alg. Disc. Meth.* 42, 1983, pg. 312-316.

[GLS88] M. Grötschel, L. Lovász and A. Schrijver, "Geometric Algorithms and Combinatorial Optimization", Springer-Verlag, Berlin, 1988, pg. 235-236.

[GP1-79] M. Grötschel and M.W Padberg, "On the symmetric travelling salesman problem I: inequalities", *Mathematical Programming* 16, 1979, pg. 265-280.

[GP2-79] M. Grötschel and M.W Padberg, "On the symmetric travelling salesman problem II: lifting theorems and facets", *Mathematical Programming* 16, 1979, pg. 281-302.

[GT91] M. Goemans and K. Talluri, "2-Change for k-Connected Networks", *Operations Research Letters* 10, 1991, pg. 113-117.

[HKRS97] M.R. Henzinger, P. Klein, S. Rao and S. Subramanian, "Faster shortest-path algorithms for planar graphs", *Journal of Computer and System Sciences* 55(1), 1997, pg. 3-23.

[JRR95] M. Jünger, G. Reinelt and G. Rinaldi, "Handbook on Operations Research and Management Science", *M.O. Ball et al.*, North Holland, 1995, pg.225-330.

[K30] K. Kuratowski, "Sur le problème des courbes gauches en topologie", *Fundamenta Mathematicae* 15, 1930, pg. 271-283.

[K97] G. Kant, "Algorithms for Drawing Planar Graphs", PhD Thesis, Universiteit Utrecht, Holland, 1993.

[L99] A. Liebers, "Planarizing Graphs - A Survey and Annotated Bibliography", Technical Report 12, Universität Konstanz, Germany, 1999, 62 pages. Available electronically at http://www.fmi.uni-konstanz.de/Preprints.

[LE00] A. Letchford, "Separating a superclass of comb inequalities in planar graphs", *Math. Oper. Res.* 25(3), 2000, pg. 443 - 454.

[MMN93] K. Mehlhorn, P. Mutzel and S. Näher, "An Implementation of the Hopcroft and Tarjan Planarity Test and Embedding Algorithm", Technical Report MPI-I-93-151, Max-Planck-Institut für Informatik, Germany, 1993, 46 pages.

[MO] http://www.densis.fee.unicamp.br/~moscato/TSPBIB_home.html

[NT98] D. Naddef and S. Thienel, "Efficient Separation Routines for the Symmetric Traveling Salesman Problem I: General Tools and Comb Separation", Technical Report, Universität zu Köln, 1998, 23 pages.

[OS94] T. Odenthal and Mark Scharbrodt, "Maximal planarization as a tool for approximating thickness and crossing number", *Student Proceedings 8th Conference of the European Consortium for Mathematics in Industry, ECMI'94*, Kaiserslautern, Germany, 1994, pg. 137-151.

[PG85] M. Padberg and M. Grötschel, "Polyhedral computations", in *The Traveling Salesman Problem* (Lawler et al., eds.), John Wiley & Sons, Chichester, 1985, pg. 307-360.

[PH80] M. Padberg and S. Hong, "On the symmetric travelling salesman problem: a computational study", *Mathematical Programming Study* 12, 1980, pg.78-107.

[PR82] M. Padberg and G. Rao, "Odd minimum cut-sets and b-matchings", *Mathematics of Operations Research* 7, 1982, pg. 67-80.

[PR91] M. Padberg and G. Rinaldi, "A branch-and-cut algorithm for the resolution of large-scale symmetric travelling salesman problems", *SIAM Rev.* 33, 1991, pg. 60-100.

[R91] K. Rosen, "Discrete Mathematics and its Applications", 2nd edition, McGraw-Hill, Inc., 1991.

[S90] W. Schnyder, "Embedding Planar Graphs on the Grid", *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'90*, 1990, pg. 138-148.

[W36] K. Wagner, "Bemerkungen zum Vierfarbenproblem", *Jber. Deutsch. Math. Verein* 46, 1936, pg. 26-32.

[WAN83] T. Watanabe, T. Ae and A. Nakamura, "On the NP-hardness of edge-deletion and -contraction problems", *Discrete Applied Mathematics* 6, 1983, pg. 63-78.