Primal Cutting Plane Methods for the Traveling Salesman Problem

by

Christos Stratopoulos

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Combinatorics and Optimization

Waterloo, Ontario, Canada, 2017

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Most serious attempts at solving the traveling salesman problem (TSP) are based on the *dual fractional* cutting plane approach, which moves from one lower bound to the next. This thesis describes methods for implementing a TSP solver based on a *primal* cutting plane approach, which moves from tour to tour with non-degenerate primal simplex pivots and so-called primal cutting planes. Primal cutting plane solution of the TSP has received scant attention in the literature; this thesis seeks to redress this gap, and its findings are threefold.

Firstly, we develop some theory around the computation of non-degenerate primal simplex pivots, relevant to general primal cutting plane computation. This theory guides highly efficient implementation choices, a sticking point in prior studies.

Secondly, we engage in a practical study of several existing primal separation algorithms for finding TSP cuts. These algorithms are all conceptually simpler, easier to implement, or asymptotically faster than their standard counterparts.

Finally, this thesis may constitute the first computational study of the work of Fleischer, Letchford, and Lodi on polynomial-time separation of simple domino parity inequalities. We discuss exact and heuristic enhancements, including a shrinking-style heuristic which makes the algorithm more suitable for application on large-scale instances.

The theoretical developments of this thesis are integrated into a branch-cut-price TSP solver which is used to obtain computational results on a variety of test instances.

Acknowledgements

Working on this thesis has been an immensely rewarding and instructive experience. I feel humbled thinking of all that I have learned and accomplished since beginning my studies here at the University of Waterloo.

First and foremost I thank my supervisor, Bill Cook, for his supportive and pragmatic guidance. Bill helped me choose an area of research which I am genuinely passionate about, one which has allowed me to develop as a mathematician and computer programmer in exactly the way I hoped for at the outset of my graduate work.

In my research I have endeavoured to build a TSP solver more or less from scratch. Bill gave me freedom to learn and make mistakes when I needed to reinvent the wheel, redirecting me when reinvention was *not* necessary but I felt an impulse to do so anyway. I feel that we struck a near-optimal balance in this regard. I arrive now at the end of my degree with deep appreciation for the genius and technical ability of all the researchers who have approached the TSP in the decades before me. I hope that my contribution can be of some use in this area, and to researchers who might wish to apply a similar approach to other problems.

I also thank and acknowledge Adam Letchford, Andrea Lodi, and Lisa Fleischer for their research related to primal cutting plane computation and TSP cuts. I of course owe a great deal to all the researchers responsible for the vast body of extant TSP literature; but specifically the work of Fleischer, Letchford, and Lodi is the basis for all of the novel research and results in my thesis. I also thank Andrea Lodi in particular for agreeing to meet with me in Fall 2016 for a discussion on my research. Andrea gave me a lot of insight into the methods and motivation of his work with Letchford and Fleischer, while also listening to some of the ideas I was exploring at the time. This talk was illuminating and encouraging, helping me to better appreciate (and do more justice to) some of the ideas I explore in Chapters 5–7 of my thesis.

I also give thanks to the people and organizations who provided financial support for my thesis research. Foremost among these was the Ontario Graduate Scholarship grant, made possible by the University of Waterloo and the Province of Ontario, and the Dr. Joseph Wai-Hung Liu Graduate Scholarship. I also thank the University of Waterloo for supplementing this grant with the President's Graduate Scholarship, and for the funding administered by the C&O graduate department. In my first three terms as a graduate student I worked as a TA marking assignments and tests, while also teaching a weekly tutorial session in the Sprint 2016 term. Teaching a classroom of students was a highly rewarding experience for me, and I learned a lot by revisiting my undergraduate course material from a teacher's perspective. (After a highly theoretical grounding in optimization, you can learn a lot by helping students work through by-hand simplex computations on 4×4 matrices!) In my final two terms of study, with additional support from Bill's funding, I was able to work on my research full time. This was an extremely productive

period for me. For months at a time I was able to devote full-time work days to nothing but writing my thesis and developing/testing my computer code. I am grateful for all this support, and I would not have been able to accomplish all that I did without it.

Thank you also to Ricardo Fukasawa and Joseph Cheriyan, for reading and reviewing my thesis. My thesis is greatly improved for having addressed your concerns and suggested revisions.

Thank you to my partner, friends, and family for your encouragement and emotional support: you all help put the "life" in "work-life balance", which is something I struggle with. Finally, thank you to the staff and crew at VIA Rail Canada.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Given a collection of cities with pairwise travel costs between them, the Traveling Salesman Problem (TSP) asks for the shortest route that visits each city exactly once, ending up where it started. The TSP is an $\mathcal{NP}$-hard combinatorial optimization problem, and indeed it is one of the most well-studied problems of this sort. Following the foundational work of Dantzig, Fulkerson, and Johnson [8], most serious attacks on the TSP have been based on the formulation of the TSP as an integer program in binary variables. This approach has been employed with great success, e.g., by Padberg and Rinaldi [24]; Jünger, Thienel, and Reinelt [14]; and Applegate, Bixby, Chvátal, and Cook [2]. All of these studies employ a *dual fractional* branch-and-cut approach, in which a linear programming (LP) relaxation is improved by repeated optimization and addition of cutting planes.

This thesis focuses on the *primal* cutting plane approach. In the primal approach we are not concerned with improving the LP relaxation as an end in itself. Rather, the LP relaxation is used as a means for improving a given starting tour, or proving that a tour is optimal. With the notable exception of Padberg and Hong's 1980 paper [25], primal cutting plane solution of the TSP has gone largely unexplored in the literature.

All the results in this thesis are based around a computer code, called *Camargue*, which implements the ideas and algorithms we shall discuss. Camargue is freely available online at

https://github.com/cstratopoulos/camargue

This thesis serves partially as documentation of Camargue, but Camargue is packaged with its own documentation as well. In particular, for every figure and table presented in this thesis, there are simple instructions for how the same experiment may be reproduced.

1.1 Formulating the TSP

In this section we will briefly review the standard integer programming formulation of the TSP, taking the opportunity to fix notation and definitions as needed.

Let G be a complete, undirected graph with vertex set V and edge set E. We let m stand for $|E|$, and n for $|V|$. For each e in E, let c_e be an integer corresponding to the travel cost for e. Note that we may regard c as a vector indexed by E, hence writing $c \in \mathbf{Z}^E$, or $c \in \mathbf{Z}^m$.

A tour T on G is a cyclic permutation of V,

$$T = (i_1, i_2, \ldots, i_n) \text{ for distinct } i_j \text{ in } V,\ 1 \leqslant j \leqslant n.$$

Any tour T induces a subgraph of G which is a Hamiltonian cycle. It has vertices V and edges

$$E(T) = \left\{ (i_j, i_{(j+1) \mod n}) : 1 \leqslant j \leqslant n \right\}.$$

The cost of the tour T is $\sum(c_e : e \in E(T))$.

For an arbitrary vector x in $\mathbf{R}^E$, and a subset F of E, we define $x(F)$ to be $\sum(x_e : e \in F)$. Using this summation notation, the cost of T is easily written as $c(E(T))$.

Given a subset S of V, write $\delta(S)$ for the set of edges with one end in S and one end not in S; this is the cut associated to S. In the case S is a singleton $\{v\}$, we just write $\delta(v)$ for the edges incident with v.

With this notation in hand, we may state a formulation of the TSP as a 0-1 integer program:

$$\begin{array}{lll} \min & c^T x & \\ \text{s. t.} & x(\delta(v)) = 2 & \text{for all } v \text{ in } V, \\ & x(\delta(S)) \geqslant 2 & \text{for all } S \text{ contained in } V,\ 2 \leqslant |S| \leqslant |V| - 2, \\ & x \in \{0,1\}^m. & \end{array}$$

The *TSP polytope* consists of all feasible solutions to this problem.

The equation $x(\delta(v)) = 2$ is called the *degree equation* for v. The inequality $x(\delta(S)) \geqslant 2$ is called the *subtour elimination constraint*, or SEC, for the set S. The formulation above is thoroughly intractible for a variety of reasons, foremost among them the fact that integer programming is $\mathcal{NP}$-hard in general.

Generally we consider an LP relaxation of the problem above, where the constraints imposed are a rapidly changing collection of cutting planes which define facets or high-dimensional faces for the TSP polytope. In light of this we may wish to employ some more vague notation. Let $P = \{x \in \mathbf{Q}_+^m : Ax \leqslant b, 0 \leqslant x \leqslant 1\}$, where $Ax \leqslant b$ are the degree equations together with

some collection of cutting planes. Moreover let P_I denote the set of all integer points in P. Then the LP relaxation may be written simply as $\min\left(c^T x : x \in P\right)$, and the IP formulation is $\min\left(c^T x : x \in P_I\right)$.

Suppose that x^* is a vector in P but not in P_I. The *standard separation problem* for x^* is to find a cutting plane $\alpha^T x \leqslant \beta$ that is valid for all vectors in P_I, but violated by x^*, hence $\alpha^T x^* > \beta$. This situation is sketched with a standard textbook illustration in Figure 1.1. Now

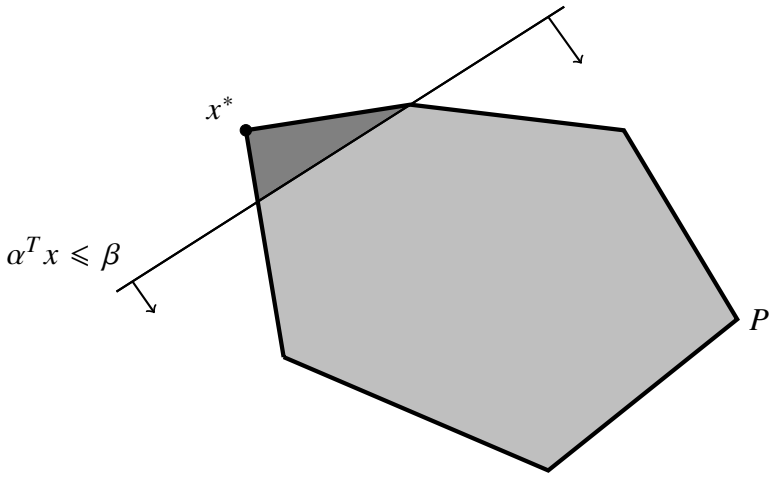


Figure 1.1: A standard cutting plane $\alpha^T x \leqslant \beta$ separating a point x^* in $P \setminus P_I$.

suppose that we also have a feasible integer solution $\bar{x}$ in P_I, along with the LP solution x^*. The *primal separation problem* for x^* and $\bar{x}$ is to find a cutting plane $\gamma^T x \leqslant \delta$ that is valid for P_I, violated by x^*, and holds at equality at $\bar{x}$, hence $\gamma^T \bar{x} = \delta$. This is sketched in Figure 1.2. This picture provides some appealing intuition about the use of primal cutting planes as well: the cut separates not only the point x^*, but the entire line segment from $\bar{x}$ to x^*. Evidently all primal cutting planes are standard cutting planes, but the reverse is not true. Moreover, if x' is another TSP tour, a primal cutting plane for x' and x^* need not be a primal cutting plane for $\bar{x}$ and x^*.

Given a class $\mathcal{F}$ of cutting planes, a *standard separation algorithm* for $\mathcal{F}$ and x^* is an algorithm which searches for inequalities of type $\mathcal{F}$ that are violated by x^*. The algorithm is *exact* if it is guaranteed to find a violated inequality of type $\mathcal{F}$ if one exists, otherwise it is called a *heuristic* algorithm. Exact and heuristic primal separation algorithms for $\mathcal{F}$, x^*, and $\bar{x}$ may be defined similarly.

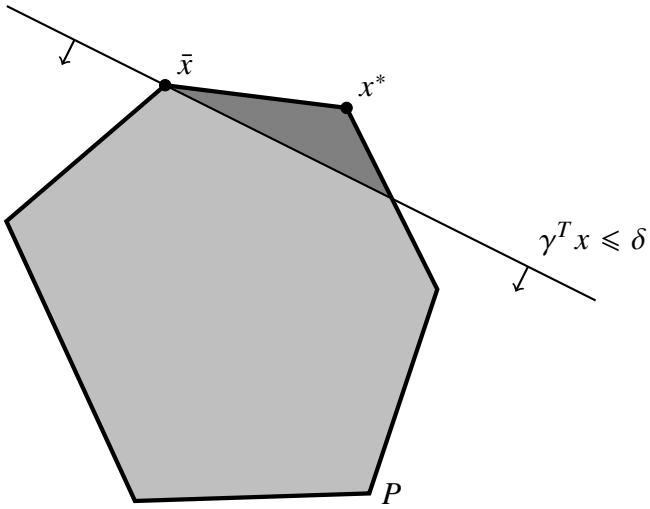


Figure 1.2: A primal cutting plane $\gamma^T x \leqslant \delta$ which is tight at $\bar{x}$ and separates x^* in $P \setminus P_I$.

1.2 Structure and Summary of this Thesis

This thesis is a study of TSP computation, serving to document the algorithms employed in the Camargue software. As such, all results and experiments are anchored, in some way or another, in TSP computation. But enduring interest in the TSP can be credited in large part to its use as an engine of discovery for general methods in discrete optimization, and this thesis pursues a development in the same spirit.

The material from Chapters 3, 6, and 7 is written with an eye towards general integer programming. The hope is that material from these chapters may be used to apply primal methods to other classes of integer programming problems, while also increasing the size of problems that may be solved with primal methods. All the other chapters are concerned more explicitly with TSP cuts and exact TSP solution, but generally there is a mixture of material. What follows is an account of topics and findings, essentially an abstract for each chapter.

Chapter 2 is an extended literature and background review. We will develop some more intuition for primal methods by emphasizing points of departure with the dual fractional case, while also reviewing primal cutting plane computational studies. The aim is to contextualize the work of this thesis, identifying difficulties encountered by previous researchers in using primal methods to attack the TSP and other IP problems.

Chapters 3 and 6 both treat linear programming theory from a primal perspective. The dual fractional approach is the dominant model for using linear programming to solve integer programs, and this is reflected both in the software available for LP computation, and in the vocabulary used to describe LP algorithms. Chapter 3 is thus concerned with developing fundamental

operations for use in primal cutting plane computation. Its main finding is a prescription for the implementation of non-degenerate primal simplex pivots, with experiments showing that dramatic speedups can be obtained as compared with approaches in existing literature.

With the machinery of Chapter 3 established, Chapter 6 treats the question of heuristics for managing LP relaxations. Namely, we consider problems of edge pricing and cut management, developing counterparts to well-known dual fractional heuristics. These methods can be used to handle LP relaxations which may accumulate a large number of rows and columns over the course of the solution process, a less pressing issue in existing computational studies.

Chapters 4 and 5 are devoted entirely to the separation of TSP cuts as implemented in the Camargue solver. Chapter 4 is more of a survey review: we discuss the implementation of primal separation algorithms for subtour cuts, blossoms, and Gomory cuts as outlined by Letchford and Lodi ([18], [20]). The discussion on SECs is straightforward, and with blossoms we describe an implementation involving shared-memory parallelism. The treatment of Gomory cuts offers a more novel synthesis of ideas, integrating primal separation of Gomory cuts into the numerical safety framework of Cook, Dash, Fukasawa, and Goycoolea [6].

Chapter 5 is the longest single chapter of this thesis, focusing entirely on the separation of simple domino-parity inequalities. We describe an implementation of the standard separation algorithm of Fleischer, Letchford, and Lodi [10], adapted to the primal case as outlined by Letchford and Lodi [20]. Moreover, we describe extensive modifications, particular to the primal case, which reduce the asymptotic complexity and storage requirements of a key phase of the algorithm. These developments are complemented with a shrinking-style heuristic which proves vital to applying the algorithm on larger-scale instances.

Chapter 7 is concerned with the integration of primal methods into a branch-and-cut search. From the point of view of general integer programming, we seek primal analogues of methods such as strong branching, best first branching, and best estimate branching, all of which are fruitfully employed in dual fractional IP methods. The question of enforcing branching subproblems is nontrivial in the primal case, and we describe an approach which is predicated on the availability of fast and highly effective tour-finding heuristics for the TSP.

The thesis concludes in Chapter 8 with a computational study of the Camargue solver, integrating all the methods of the previous chapters. We judge the efficacy of Camargue as an exact TSP solver, using experiments with random Euclidean instances and TSPLIB [28] problems. Although Camargue is not competitive with the Concorde solver of Applegate et al. [2], it is indeed capable of solving small- and medium-sized TSP instances, with exact solutions reported for instances of size around 1,000 nodes. Moreover, we report limited experiments on larger TSPLIB instances, quantifying the ability of Camargue to find and improve TSP tours.

Chapter 2

Primal Cutting Plane Methods

Very broadly speaking, cutting plane methods are a class of integer programming solution protocols in which an LP relaxation of an integer programming (IP) problem is tightened and improved through the addition of cutting planes. This highly simplistic viewpoint belies the many different subclasses of cutting plane methods which can be employed, and their unique characteristics.

In the following sections, we focus our attention on two paradigms for cutting plane methods. First is the *dual fractional* approach, which is the dominant paradigm in computational solution of the TSP. This material is intended partially as a review, and partially to motivate the *primal* approach, which is the main subject of this thesis. Finally, this chapter concludes with a brief literature review of primal cutting plane methods, with an emphasis on the TSP.

2.1 The Dual Fractional Approach

The dominant paradigm for the solution of the TSP is that of *dual fractional* cutting plane methods, and of dual fractional *branch-and-cut* methods. We will provide a brief overview rather than an algorithmic discussion. For details, specifically in the context of the TSP, see e.g., Applegate et al. [2], Padberg and Rinaldi [24], or Jünger et al. [14].

The process begins with the computation of a (hopefully good, or possibly optimal) tour by means of some heuristic. (Usually this is some variant of the Lin-Kernighan heuristc. See e.g., [21], [12], or Chapter 15 of [2].) The cost of this tour is then stored for later use.

A starting LP relaxation is also constructed, generally the degree LP. This starting LP is optimized, and in general its solution will be a fractional vector or integral subtour vector with cost considerably lower than the upper bound provided by the tour. Standard separation routines

are then employed to find cutting planes which cut off the current optimal LP solution, and then the dual simplex algorithm is used to optimize the new LP, hopefully netting an increase in the LP lower bound. Cutting planes are computed through a mix of heuristic and exact separation routines. This process is repeated, perhaps eventually leading to an integral tour as the optimal solution to the LP relaxation, in which case the instance has been solved.

If the instance is not solved thusly, or the progress in closing the gap becomes agonizingly slow, *branching* is employed: fractional variables are chosen to be fixed to zero or one values, measuring the resulting change in the optimal LP objective value, all the while repeating the cut generation procedure as above. With an edge fixed to a certain value, it may be the case that the optimal objective value is greater or equal to the cost of the optimal tour, in which case further branching is not necessary. It may also be the case that fixing a variable renders the LP infeasible, in which case the search may be terminated as well. In this way, one seeks by implicit enumeration to arrive at an optimal tour.

As documented in the sources above, the dual fractional branch-and-cut approach has been enormously successful in the solution of TSPs of various scales. In light of the subject of this thesis, however, let us make a series of seemingly pigeonholed or obtuse observations about this approach:

- After an initial tour is computed, its use in the solution process is reduced to a numerical upper bound, discarding its structure. [1]
- The focus is on improving the lower bound of a usually fractional solution. The discovery of an improved tour occurs rarely, perhaps well into the branching process, if at all.
- For some classes of cutting planes, exact standard separation routines may not be efficient enough for regular use.

2.2 The Primal Approach

As alluded to in the previous section, the main object of interest in the dual fractional approach is an optimal solution to the LP relaxation: we focus our efforts on improving this optimal solution, adding cutting planes and re-optimizing by use of the dual simplex algorithm.

Conversely, *primal cutting plane methods* focus their attention on a primal feasible solution to the integer programming problem, i.e., on a given TSP tour. Given that the starting tour may

[1] A notable exception is the approach of Applegate et al. [2], where the starting tour is used for heuristic SEC generation, and for compressed storage of cuts. We will revisit these ideas in Chapters 4 and 6.

be optimal or near optimal, we wish, intuitively, to stay close to the tour on the TSP polytope, adding cutting planes which either prove the optimality of the tour or lead to the discovery of a better one.

With this in mind, let us consider a simple primal cutting plane algorithm. This algorithm is a *pure* primal cutting plane algorithm in the sense that it does not contain a branching scheme of any kind. Note that, as in the dual fractional case, starting tours are generally computed using some variant of the Lin-Kernighan heuristic.

Algorithm 2.2.1: Simple Primal Cutting Plane Algorithm

Data: A TSP instance
Result: An optimal tour upon successful exit
Find a starting tour $\bar{x}$
Set the LP relaxation feasible region P to the degree LP
Construct a basis associated to $\bar{x}$
while $\bar{x}$ *is not dual feasible* **do**
    Perform a primal simplex pivot from $\bar{x}$ to a new LP solution x^*
    if $x^* \neq \bar{x}$ **then**
        if x^* *is an improved tour* **then**
            Augment: $\bar{x} \leftarrow x^*$
        else
            for *known classes* $\mathcal{F}$ *of inequalities* **do**
                Separate: call primal separation for $(x^*, \bar{x}, \mathcal{F})$
                $\mathcal{L} \leftarrow$ list of inequalities found
            if $\mathcal{L} \neq \emptyset$ **then**
                pivot back to $\bar{x}$
                $P \leftarrow P \cup \mathcal{L}$
            else
                exit `FAILURE`
exit `SUCCESS`

This simple algorithm resembles quite closely the general primal IP framework laid out by Letchford and Lodi [18], and the primal cutting plane TSP algorithm described by Padberg and Hong [25]. In the next section, we shall review the work of all these authors in more detail. As we enter a more in-depth discussion, the machinery involved will quickly grow beyond the scope of an easy representation like Algorithm 2.2.1. Nevertheless, it is a useful starting point for later algorithmic development, and to give a simple idea of what the primal cutting plane solution process looks like.

2.3 Primal Cutting Plane Literature Review

This section will consist of a brief literature review of primal cutting plane methods, with an obvious bias towards the TSP. This review is not intended to be exhaustive; rather it is meant to temper my claims about the dearth of primal cutting plane TSP research. We shall summarize the work and findings of certain papers and authors, trying to identify hints and stumbling blocks found along the way.

2.3.1 Dantzig, Fulkerson, and Johnson

The work of Dantzig, Fulkerson, and Johnson [8] may well be cited as the prototype for virtually all dual fractional branch-and-cut approaches to the TSP, and perhaps more generally as the prototype for solving arbitrary combinatorial optimization problems through LP-theoretic means. In [8] the researchers certify the optimality of a tour of length 699 through 49 cities in the United States. After the generation of several SECs and comb inequalities, the starting tour is proved optimal through certain arguments which bear a striking resemblance to branch-and-bound computations, although the concept had not been formally named as such.

What is less well-known, and indeed has perhaps been relegated to the place of historical trivia, is that Dantzig, Fulkerson, and Johnson did not solve their LP relaxations to optimality. Rather, they considered their starting tour as an extreme point of the TSP polytope, and carried out a single iteration of the simplex algorithm, a pivot, to arrive at an adjacent vertex of the TSP polytope. This adjacent vertex was then cut off by primal cutting planes. In Section 3.3 of [2], Applegate et al. give a more thorough discussion of the nature and motivation of the computational decisions made by Dantzig, Fulkerson, and Johnson. In particular, it is speculated that the difficulty of computing simplex iterations by hand may have been a driving factor for preferring single pivots to a full course of the simplex algorithm.

In the previous chapter, and in particular with the illustration of Figure 1.2, we hinted at the utility of primal cutting planes as compared with standard cutting planes. Applegate et al. expound on this motivation in the context of the TSP, and the work of Dantzig, Fulkerson, and Johnson.

> If [a tour] $\bar{x}$ is not optimal, then the simplex algorithm selects a direction to move to a new basic solution x' via a pivot operation. If x' is not a tour, then a cutting plane is added to the LP. In choosing such a cut, Dantzig, Fulkerson, and Johnson restrict their attention to cuts that are both violated by x' and hold as an equation for $\bar{x}$; this extra condition allows them to cut off the entire line segment from $\bar{x}$ to x' and thus force the simplex algorithm to move in a different direction in the next step.

As compared to the dual fractional approach, this insistence on tight cutting planes to redirect subsequent pivots has deep consequences on the interplay between cut generation and linear programming computation. The following metaphor is quite heavy-handed, but it may give some nice intuition.

Let us step back and consider the very nature of linear programming methods as a tool for solving the TSP. The LP relaxation is initialized with an extremely vague description of the problem, and some LP solver is used to compute a feasible LP solution. When this feasible solution is not a tour, cut finders are used to update the description of the LP relaxation, calling the LP solver anew. So watching the cutting plane solution process is like watching a boss assign errands to a lackey who will set upon any task with great speed, but will often misinterpret the boss's instructions. Say the boss asks the lackey to fetch an object, while not mentioning that it is in the very same room in which the two of them are standing. In the dual fractional case the lackey races out of the room, out of the building, and halfway across town, at which point the boss says "You should not have taken the elevator downstairs, and once getting downstairs you should not have left the building, and once leaving the building you should not have crossed the street, and . . . please try again." In the primal case, the boss sees that the lackey is leaving the room and says "Stop! You should not leave this room. Please try again."

2.3.2 Padberg and Hong

To my knowledge, no study since the work of Padberg and Hong [25] has undertaken computational solution of the TSP by primal cutting plane methods. Their paper provides an algorithmic overview of a pure primal cutting plane algorithm, i.e., one with no branching scheme of any kind. The paper is a thorough theoretical and computational study whose influence can still be seen in modern TSP computation.

Padberg and Hong use starting tours computed by the Lin-Kernighan heuristic [21], describing a simple procedure from which one may obtain a starting basis for the simplex method. Then, they choose to pivot to adjacent vertices using their own implementation of the steepest edge criterion for primal simplex pivots, writing that "it appears to be difficult to identify improving tour vertices," so that one either pivots to a new LP solution or carries out a degenerate pivot which effects no change on the solution vector.

In the event that a non-degenerate, non-tour pivot is identified, Padberg and Hong proceed with their constraint identification procedures, which are in fact primal separation algorithms. They consider inequalities arising from subtours, blossoms, and combs, all of which are now recognized as invaluable, facet-defining TSP cutting planes. Another class of inequalities considered is the class of *chain* constraints, which define high-dimensional faces rather than facets; these seem to have fallen out of use in TSP computation.

Padberg and Hong describe a polynomial time exact primal separation routine for subtour inequalities, which is simpler and faster than the minimum cut approach used in the standard case. For the remaining classes of inequalities, they resort to heuristic separation methods.

In the separation of blossoms, Padberg and Hong employ heuristics based on odd components and blocks of the support graph. If a violated inequality is found thusly, they simply check whether it also holds at equality at the current tour. Thus, their primal separation heuristic is just a standard separation heuristic with an extra step at the end. And indeed, Applegate et al. (see Section 7.1 of [2]) use Padberg and Hong's so-called odd component heuristic for standard separation of blossom inequalities. Naturally, since they are solving the standard separation problem, there is no need for them to check tightness of the cut at a given tour.

This suggests an interesting interplay between primal and standard separation algorithms. In the example just given, a standard separation heuristic is turned into a primal separation heuristic with little added effort. The converse may also take place: in Section 6.3 of [2], Applegate et al. describe how to use primal SECs as a fast heuristic for standard separation. This will be discussed in much more detail in Section 4.1.1.

The final step of Padberg and Hong's algorithm is handling the case where no primal cutting planes are identified. As mentioned, theirs is a pure primal cutting plane algorithm with no branching. Thus, in the event that no violated primal cuts are found, they solve the current LP relaxation to optimality. The optimal solution provides a lower bound on the cost of an optimal tour, and their existing tour provides an upper bound, thereby giving a bound on the optimality gap.

2.3.3 Letchford and Lodi

We close this literature review with an account of the work of Adam N. Letchford and Andrea Lodi on primal separation algorithms and primal cutting plane methods. Although their work is written from a more general perspective of mixed integer programming, it contains many valuable insights for primal cutting plane solution of the TSP, as well as some descriptions of exact primal separation algorithms for the TSP.

Also, having been written throughout the 2000s, the work of Letchford and Lodi is much more contemporary than any other sources available. As we mentioned, the work of Padberg and Hong predates the existence of widely available linear programming codes. And with Dantzig, Fulkerson, and Johnson the situation is even more stark, as they solved their LP relaxations using by-hand computations. Thus, from a computational point of view, the work of Letchford and Lodi is an invaluable starting point.

In 2003, Letchford and Lodi [20] released a paper on primal separation algorithms in integer programming. The authors engage in a theoretical study of some complexity aspects of the primal

separation problem, together with a description of separation algorithms for various specific classes of integer programming problems. Most interesting from the point of view of this thesis, it contains a description of exact primal separation routines for subtour inequalities, blossom inequalities, and simple domino parity (DP) inequalities. [2] In the case of subtour inequalities and blossom inequalities, the algorithms are faster than their standard counterparts by a polynomial factor. No asymptotic speedup is claimed for the separation of simple DP inequalities, but they describe a dramatic conceptual simplification of a key phase of the algorithm. Fleischer et al. claim a polynomial runtime in [10] which is partially predicated on a state-of-the-art deterministic minimum cut algorithm. Letchford and Lodi [20] describe this algorithm as "rather tricky to implement", before going on to say that in the primal case this step is rendered "almost trivial", as it can be replaced with a naive, enumerative approach.

Letchford and Lodi's 2002 study [18] is written at a more general level, providing a simple algorithmic framework for a modern primal cutting plane algorithm. The suggested approach involves the use of facet-defining primal cuts, as well as general-purpose mixed integer cuts, such as Gomory cuts, which are chosen to solve the primal separation problem. From the point of view of the TSP, they also describe a means for handling the situation where one pivots to an integral solution which is not a tour but also does not violate any primal cutting planes. Such a situation can arise in the TSP when an integral subtour vector does not violate any primal SECs. We treat this situation in detail in Section 4.1.2.

Letchford and Lodi also consider briefly the possibility of branching in the primal context, mentioning the difficulty of adapting dual fractional branching protocols which would destroy the feasibility of a given tour. That is, branching on a variable x_i is traditionally performed by imposing a constraint $x_i = 0$ on one branch and $x_i = 1$ on the other. But in the primal case we are using some tour $\bar{x}$ as a vertex to pivot from, and to generate cutting planes; one of $x_i = 0$ and $x_i = 1$ inevitably renders $\bar{x}$ infeasible.

Letchford and Lodi explore this idea more thoroughly in [19], laying out an algorithmic framework for embedding primal cutting plane methods in a branch-and-cut procedure. This is named Augment-and-Branch-and-Cut, which may be pleasantly abbreviated to ABC. In a standard dual fractional branch-and-cut search, no tour is feasible on two children of the same branching node. In the primal case, the next best thing would be to make it so that only one tour is feasible on both branches; this is what the authors accomplish with their so-called "Main Branching Rule". The rule works by adding a collection of rows, each having two nonzero entries,

[2] At the time of publication, Letchford and Lodi had released a conference paper [17] in which they claimed to have found a polynomial-time exact separation algorithm for the class of simple comb inequalities. The full version of this paper [10] was released in 2006 with Lisa K. Fleischer as an added author. In this paper, the authors corrected the results in [17], showing that their algorithm finds a proper superclass of simple comb inequalities, the simple domino parity inequalities. Thus, the paper [20] still refers to a separation algorithm for simple comb inequalities, but the results quoted here are not affected in any substantial way.

to the LP, where the number of rows added is on the order of the number of columns in the LP relaxation. We review this work in more detail in Chapter 7, considering difficulties that may arise in adapting this protocol to TSP computation.

An interesting feature of Letchford and Lodi's work is the computational experiments, which are carried out using the commercial IBM ILOG CPLEX code. To perform primal pivots, the authors invoke the primal optimization function of CPLEX with the iteration limit artificially restricted to one pivot. (Andrea Lodi, 2016, private communication.) After each pivot, they check, by comparing objective values, whether a non-degenerate pivot has been identified. In the computational results of the ABC paper [19], they describe tests with highly degenerate, challenging integer programs, in which it is stated that "more than 95% of the time is spent performing degenerate pivots". Indeed, commercial LP codes employ a host of measures for dealing with degeneracy and stalling, and these can be employed *during* the optimization process if the iteration count or computation time seems poor. However, it would appear that choking the solution process after one pivot destroys access to these measures. In Section 3.3 we set upon empirically validating this speculation, while also providing novel, alternative approaches.

2.3.4 Concluding Remarks and Retrospective

The preceding discussion brings us to an interesting challenge for the practical implementation of primal cutting plane methods. Namely, the development of commercial LP codes is tied very closely to the development of solution methods for the TSP (see Chapter 13 of [2] for a thorough discussion), and the dominance of the dual fractional approach is thusly reflected in the facilities provided by commercial LP solvers.

At some point in the late 1970s or early 1980s, when Padberg and Hong published their study [25], the future of TSP computation seems still to have been at a nascent, undetermined stage. Their study was ambitious and fairly successful, and the authors wrote that their results "lend convincing support to the hypothesis that inequalities defining facets of the convex hull of tours are of substantial computational value". One also might remark that their work provides convincing support for the computational value of the primal approach more generally. The year 1985 saw the first publication of the classic book *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, and at this point in time the matter appears to have been settled. Padberg himself, along with Martin Grötschel, contributed a chapter [23] to this book, which contains an extensive discussion of separation algorithms as well as computational results. The authors provide some remarks and a bit of a technological state-of-the-art review to justify their computational choices.

> Mathematical programming software products, such as CDC's code APEX,

> IBM's software system MPSX-MIP/370, Ketron's MPSIII, SCICON's code SCICONIX, Sperry Univac's FMPS, etc., are all highly developed products, and are the outcome of years of effort that combined the skills of many mathematicians, computer scientists and computer programmers. It is therefore not only natural but also advisable to use such software systems as building blocks for the practical solution of difficult combinatorial optimization problems. More precisely . . . we propose to use the *most efficient* linear programming solver that is available.
>
> This implies in particular that we favor a *dual* cutting plane approach over the *primal* cutting plane approach, though this is partly due to the fact that a primal cutting plane approach requires a different set of linear programming tools than are available.

Later the authors make the following remarks, essentially reviewing their own previous work.

> In a computational study [Padberg & Hong, 1980] a primal cutting plane approach was employed and this required the writing of a simplex code of their own. In more computational studies [Grötschel, 1977a, 1980b; Crowder & Padberg, 1980; Crowder, Johnson & Padberg, 1983] a dual cutting plane approach was adopted and IBM's system MPSX/370 was used to solve the linear programs. Based on this experience we prefer the dual cutting plane approach, last but not least since the dual approach permits one to treat the linear programming routine as a 'black box' in combinatorial problem solving.

Thus it would be inaccurate to attribute the dominance of dual fractional solution protocols to technological convenience alone, as it was grounded quite clearly in the success of several computational studies. Nonetheless, this provides interesting historical motivation, and we shall use the words of Padberg and Grötschel to define a wish-list for the implementation of primal cutting plane machinery. Namely, the linear programming solver must be a black box in computation, and it should be put to use in the most efficient way possible. These extremely simple criteria will be used to ground and motivate the discussion of the following chapter.

Chapter 3

Fundamental Operations on Linear Programs

To take the simplest possible view, solution of the TSP by dual fractional cutting plane method goes like this: we optimize via dual simplex algorithm, add cuts, and optimize again. From this point of view, dual optimization is the driving force behind the solution process. That is, we rely on repeated optimization to reveal new sources of cutting planes, and we rely on cutting planes to redirect the optimization. The interplay between these two is therefore a core, fundamental component of the solution process.

Let us define the following notion: A *fundamental LP-theoretic operation* is a means by which we interact with LP relaxations to advance and redirect the solution process. Thus in the dual fractional case, the fundamental operations are dual optimizing and adding cuts: we optimize the LP to get to a new LP solution for which we can call separation routines. And a third consideration implicit in this discussion is the construction of dual feasible bases.

To be sure, this is ad-hoc, non-standard terminology, and one might convincingly argue that we won't advance the solution process very far without operations for pruning cuts, branching, and pricing edges. Nevertheless, let us focus on this definition for the time being: our goal is to establish fundamental LP-theoretic operations for primal cutting plane methods, emphasizing situations where they depart from the dual case.

The process of adding cutting planes is unchanged. Rather, it is the type of cutting planes which is different in the primal case; this is the subject of the next two chapters. Thus, the topics at hand are basis construction, and finding new LP solutions for which separation routines can be called. We regard these two as direct analogues of dual basis construction and dual simplex optimization, respectively. A novel operation in the primal case is a *pivot back* operation. In Algorithm 2.2.1 we saw that one must repeatedly pivot back to the incumbent best tour, and it

turns out that approaches to this operation are closely tied to the means by which we move to new LP solutions.

3.1 Construction of an Initial Basis

In both the dual fractional and primal cases, the solution process is generally started with an LP relaxation which consists of nothing more than lower and upper bounds on variables, and the degree equation for each node in the graph. In the dual fractional case, a starting, dual-feasible basis is readily obtained by optimizing the LP via the dual simplex algorithm.

Another approach is possible, however: in Chapter 12, Section 1.3 of [2], Applegate et al. note that an optimal solution to the degree LP corresponds precisely to an optimal solution of the fractional 2-matching problem, which can be found somewhat more quickly through a polyhedral combinatorial algorithm which also constructs a starting basis.

The situation is quite different in the primal case, which calls for a primal feasible basis associated to the starting tour $\bar{x}$. That is, although the starting LP is still the degree LP, optimal solutions to the degree LP will be all but useless in the construction of bases for $\bar{x}$.

In [3], Behle et al. study primal cutting plane methods for the degree-constrained minimum spanning tree problem. They provide a procedure for primal feasible basis construction which in fact can be generalized to arbitrary 0-1 integer minimization problems. Suppose P is the feasible region of the LP relaxation, and the linear program is $\min(c^T x : x \in P)$. Then temporarily introduce an objective function c', such that c'_i is $1 - 2\bar{x}_i$. Then $\bar{x}$ is the optimal solution to $\min\left((c')^T x : x \in P\right)$, and optimizing via the primal simplex algorithm gives a primal feasible basis associated to $\bar{x}$.

Just as in the dual fractional case, though, it is possible to obtain the same result for considerably less work by considering combinatorial structure of the problem. The following approach, due to Padberg and Hong [25], is elegantly simple and certainly much more efficient than optimizing an LP or computing a fractional 2-matching.

Suppose $\bar{x}$ corresponds to the tour $(i_1, i_2, \ldots, i_n)$, and that the feasible region of the degree LP is given by the linear system $Ax = b$. Then, directly from Padberg and Hong,

> [s]ince the columns of A that are in the tour yield a submatrix B of A with row and column sums equal to two, we have a starting basis if n is odd. If n is even, B is singular. In this case we modify B by discarding the column corresponding to the edge (i_{n-1}, i_n) and introducing in its place the edge (i_1, i_{n-1}) ... (Of course, the discarded column (i_{n-1}, i_n) is marked to be nonbasic and at its upper bound of 1.)

This Padberg-Hong approach is used to construct starting tour bases in Camargue, with only a minor change. Almost all of Padberg and Hong's studies are performed with the complete graph edge set, but in Camargue we opt for a more sparse selection of starting edges. (This matter is discussed explicitly in Chapter 6.) If the edge (i_1, i_{n-1}) required by the Padberg-Hong procedure is not present in the initial edge set, it is added manually. An artificial, extremely large positive travel cost is assigned to the edge in the event that it is not present in a sparse instance under consideration.

3.2 Moving to and from LP Solutions

The next operation to be adapted to the primal case is that of optimization. In a given step of a dual fractional algorithm, we may have some non-tour LP solution x^* which is separated by cutting planes. The new LP is optimized to obtain some x^{**}, which may be an optimal tour, or may have objective value that proves the optimality of a given tour. If not, x^{**} is a source for the generation of further cutting planes, so the solution process can progress. Either way the goal is to bring an optimal LP solution closer and closer to an optimal tour. In this view, it makes sense to repeatedly optimize the LP after adding rounds of cutting planes.

Recall, however, that the primal approach aims to augment, or prove the optimality of, a given starting tour $\bar{x}$. This starting tour will generally be computed by some variant of the Lin-Kernighan heuristic, and tours obtained thusly are often optimal or near-optimal. Thus, intuitively, we wish to stay close to $\bar{x}$ on the TSP polytope.

But this is only motivation for why we do not optimize the LP relaxation at every step; we have not yet established a primal counterpart for dual optimization.

Let us consider the approach adopted by Letchford and Lodi in [18]. Their algorithmic overview starts with constructing a primal feasible basis for the initial solution $\bar{x}$. Then they write:

- Step 2: If the basis for $\bar{x}$ is dual feasible, stop.
- Step 3: Perform a primal simplex pivot. If it is degenerate, return to 2.

The corresponding parts of the algorithm used by Padberg and Hong [25] are nearly identical.

As discussed in the previous chapter, in both papers the authors remark that the computation of degenerate pivots is a considerable computational expense. Indeed, degenerate pivots seem to be the rule rather than the exception when solving LP relaxations of the TSP. (See Chapter 13, Sections 2–4 of [2] for a thorough discussion.)

For this reason, we do not declare "primal simplex pivot" to be a fundamental operation on LP relaxations arising in primal cutting plane solution of the TSP. We want a dual feasible basis for $\bar{x}$, or a new solution x^* that is either an improved tour, or from which we can generate primal cutting planes. And, we want to stay close to $\bar{x}$ on the TSP polytope. A single primal simplex pivot does the latter, but fails at the former more often than not. And indeed, "dual simplex pivot" is not the fundamental operation for moving to new LP solutions in the dual fractional case either. Rather, optimizing via dual simplex algorithm consists of many (mostly degenerate) dual simplex pivots.

With the preceding motivation, a definition is in order. Let $\bar{x}$ be a tour, and let B be a primal feasible basis for $\bar{x}$. Then a *primal non-degenerate pivot* is an operation which takes $\bar{x}$ and B as input, and returns a primal feasible basis B' such that B' is either

- a dual feasible basis for $\bar{x}$, proving the optimality of $\bar{x}$; OR
- a basis for some x^* distinct from $\bar{x}$, and such that there exists a basis B_0 for $\bar{x}$ which differs from B' by a single primal simplex pivot.

What exactly is asserted by this definition? Consider optimizing an LP relaxation by a call to the primal simplex algorithm, having provided the solution-basis pair $(\bar{x}, B)$ as a starting basis. And let $(\hat{x}, \hat{B})$ denote an optimal primal solution, with primal- and dual-feasible basis $\hat{B}$. The primal simplex algorithm proceeds at each iteration by a pivot step, choosing a pair of entering and leaving variables. Thus, starting with the basis B, we obtain a sequence of bases

$$B, B_1, B_2, B_3, \ldots, B_k = \hat{B},$$

such that B_1 differs from B by a single pivot, and likewise for each B_i and B_{i-1} for i bigger than one. But although the bases are distinct, primal degeneracy means that the resident solution may remain stagnant for a while. Thus, for some j, usually less than k; and some x^*, usually distinct from $\hat{x}$, a subsequence of the solution-basis pairs will look like

$$(\bar{x}, B), (\bar{x}, B_1), (\bar{x}, B_2), \ldots, (\bar{x}, B_{j-1}), (x^*, B_j).$$

Thus for many steps we just get a new primal feasible basis for $\bar{x}$, until finally arriving at a basis B_j for a distinct vector x^*. This (x^*, B_j) is what ought to be returned by a non-degenerate pivot. Thus B_j is the basis referred to as B' in the definition above. As for the basis B_0 in that notation, it is a basis for $\bar{x}$ which differs from B' by a single primal simplex pivot. If $B' = B_j$, that means B_0 is B_{j-1}. This describes the case of moving from $\bar{x}$ to a distinct new tour or non-tour vector x^*. In the other situation, $\bar{x}$ is equal to the optimal solution $\hat{x}$, and $\hat{B}$ is the basis for $\bar{x}$ that proves its optimality.

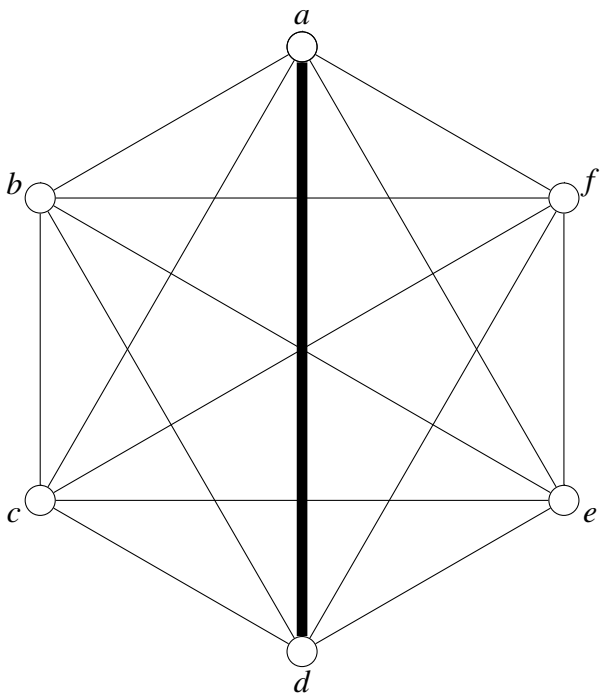


Figure 3.1: A weighted complete graph on six vertices. The thick line has weight three, all others have weight one.

At this point we have developed "non-degenerate pivot" as an abstract operation, departing from the approach adopted in existing primal cutting plane literature. We now seek to characterize this departure more explicitly, thereby showing what can be gained from it. Consider the following two instructions:

- Instruction 1: Repeatedly compute single primal simplex pivots until one is non-degenerate.
- Instruction 2: Compute a non-degenerate primal simplex pivot.

At a first glance the distinction between the two seems like pure semantics, since both instructions compute the same end result. The key point is that Instruction 2 specifies an abstract, atomic operation. Instruction 1 is a possible *implementation* of Instruction 2, but others are possible, as we shall see in the remaining sections of this chapter.

We now consider an analogy to path-finding problems from graph theory, aided by Figure 3.1. Let G be the graph in Figure 3.1, and suppose we want to find a path from a to d in G. The following algorithm is perfectly valid. Mark all nodes of the graph unvisited, except for a. Let x

be an unvisited neighbour of a such that the cost of (a, x) is minimal among all possible unvisited neighbours of a. If x is d, we are done. If not, mark x as visited and repeat from x, examining its unvisited neighbour set. Repeat until finding d.

The correctness of this algorithm is readily apparent, as is its woeful inefficiency. In the graph illustrated in Figure 3.1, it finds the *longest* path from a to d, both in terms of total edge cost and number of intermediate vertices. And this situation is made all the more absurd given that a and d are *adjacent* in G. But "find a path from a to d" is an abstract request, making no particular insistence on efficiency or cost. A path from a to d could just as well be computed by a shortest path algorithm, or something to that effect.

In the example just described, weights on the graph can be thought of as the score or price metrics imposed by a given pivot selection rule, such as minimum reduced cost. Performing repeated single pivots in a greedy fashion is like using the inefficient algorithm described in the previous section: although we may at each step be choosing the edge with most negative reduced cost, we keep no record of the progress or duration of the search, leaving no opportunities for redirection after a stagnant sequence of pivots. And indeed, we described a bit of an insidious situation in the example above, depicting adjacent vertices a and d that are joined by a relatively expensive edge. But this is the exact nature of degeneracy in TSP computation: the pivot vertex we are trying to identify is *adjacent* to the current tour on the TSP polytope, but this information is somehow obfuscated from the point of view of our pivot selection rules. Choosing a different pivot selection rule, like steepest edge, is like assigning different edge costs in the example above: the greedy approach may then find a path from a to d with fewer intermediate vertices, but often the algorithm will still fail to "realize" that a and d were adjacent all along.

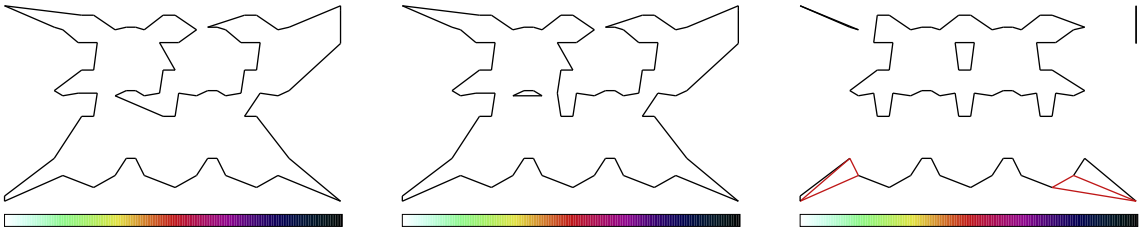
Primal degeneracy is a hard obstacle to overcome, and it does not become easier just because we give it a new name. Rather, we seek a more abstract definition to allow freedom of implementation. Instruction 1 above names an end, but it also locks us into a specific means to that end. Conversely, Instruction 2 just says what we want, without specifying how we are going to find it.

In the primal case, we now adopt the convention that

primal non-degenerate pivots should be regarded as the fundamental operation for moving to new tours, bases, or LP solutions.

In light of this, an effective implementation of a primal non-degenerate pivot operation is of foremost practical concern in implementing a primal cutting plane TSP solver.

Once we have pivoted to a new LP solution, how do we get back? This is the substance of a properly implemented *pivot back* operation, which turns out to be intimately related to the choice of non-degenerate pivot operation. Dwelling a bit longer on the notation introduced in the exposition above, suppose we started with $(\bar{x}, B)$ and computed a non-degenerate pivot to



(a) Optimal tour (b) Non-degenerate pivot (c) Optimal fractional 2-matching

Figure 3.2: Comparison of tour, single pivot, and optimal fractional 2-matching for pr76

(x^*, B_j). When it comes time to pivot back to $\bar{x}$, should we pivot back to $(\bar{x}, B)$, or $(\bar{x}, B_{j-1})$? And if we want to pivot back to $(\bar{x}, B_{j-1})$, will we even have access to the basis B_{j-1}? This is a slightly subtle matter which will be discussed in due course in the next section.

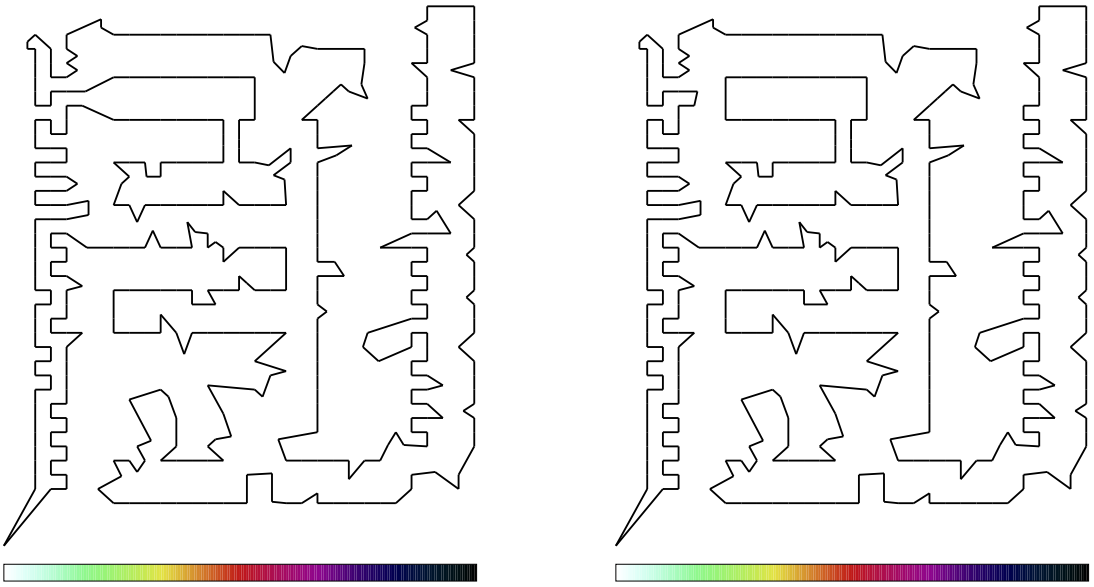
Before that, we close this section by trying to develop a bit more intuition on the nature of non-degenerate pivots. We will look at some pictures.

Figure 3.2 compares a non-degenerate primal pivot solution to an optimal dual simplex solution, using the TSPLIB [28] instance pr76 as an example. In all subfigures, the operating LP relaxation is the degree LP, with a core edge set consisting of the union of 10 chained Lin-Kernighan tours. Figure 3.2a shows an optimal tour on this instance, followed by a single non-degenerate primal pivot in Figure 3.2b. The two share considerable structure, and one easily spots a single violated connected component subtour inequality in 3.2b. On the other hand, Figure 3.2c shows the optimal solution to the degree LP obtained by the simplex algorithm. A great many more violated cutting planes are evident, but Figure 3.2c bears only passing resemblance to Figure 3.2b.

The picture painted by Figure 3.2 is intuitively appealing, but it should not be regarded as the canonical representation of a single non-degenerate pivot. The figure was obtained by performing a single non-degenerate pivot from an optimal tour to a new solution with just the degree LP as the active constraint set. At a more advanced stage in the solution process, with more cuts in the relaxation, primal pivot vectors may look a bit more complex.

Figure 3.3 again compares an optimal tour with a non-degenerate pivot obtained with no active constraints but the degree equations. This time we consider the TSPLIB [28] instance pcb442. Much like in Figure 3.2 with pr76, the pivot in Figure 3.3b is quite similar to the tour in Figure 3.3a, with a connected component SEC readily apparent.

In Figure 3.4, we consider an LP relaxation for pcb442 obtained after a sparse, pure primal cutting plane run of Camargue, with a selection of the separation routines described in the next chapter. Figure 3.4a shows a fractional non-degenerate pivot for which no further primal cuts were found. In Figure 3.4b, we show the fractional solution that results from optimizing the LP



(a) Optimal tour.

(b) Single pivot.

Figure 3.3: Comparing a tour and single pivot for pcb442 with the degree LP.

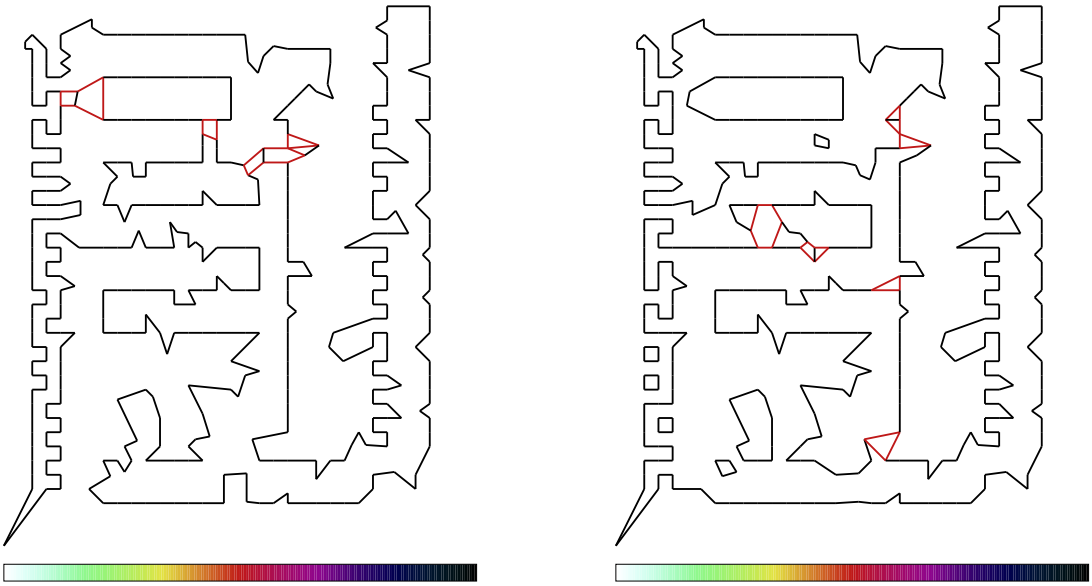
relaxation with the primal simplex algorithm. Comparing the pivot in Figure 3.4a and the optimal solution in Figure 3.4b, one might still argue that the pivot bears more resemblance to the starting optimal tour from Figure 3.3a. But certainly the difference between pivot and optimal solution is marginal at this point.

3.3 Implementing Pivot Operations

Having named some concepts and laid a bit of theoretical groundwork, we now consider strategies for computational implementation. As discussed in Section 2.3, this topic is highly dependent on existing linear programming software. At the time of writing, I am not aware of any LP solvers which provide a non-degenerate pivot functionality out of the box; this section would be moot otherwise. Thus, a more verbose title for this section could be

Efficiently Implementing Pivot Operations Using The Facilities of Dual-Fractional-Based LP Solvers.

Despite the fact that this material is highly software-specific, the goal of this section is to provide generally applicable principles, rather than implementation instructions for working with



(a) Final pivot

(b) Optimal solution after final pivot

Figure 3.4: Comparing a non-degenerate pivot and optimal LP solution for pcb442 with a run of cutting planes

a particular piece of software. Essentially, the hypothesis proposed by this section follows the discussion from the previous section with Figure 3.1: non-degenerate pivot operations should not be thought of nor implemented as repeated single primal pivots; rather they should be regarded as a truncated course of primal simplex optimization, and implemented accordingly.

Algorithmic subroutines will be described in terms of some abstract, fundamental operations, treating the LP solver as a black box. The solver is of course expected to provide the following basic functionalities:

PRIMAL-OPT	Optimize the LP with the primal simplex algorithm.
SOLUTION	Return the resident primal LP solution.
OBJVAL	Return the objective value corresponding to the SOLUTION vector.
BASIS	Return the basis corresponding to the SOLUTION.

In the remainder of this section we discuss theoretical choices and describe implementation approaches, defining additional abstract operations as needed.

3.3.1 Theoretical Considerations

Let us highlight some salient features which would guide implementation choices. Fix again some notation used to define non-degenerate pivots: suppose $\bar{x}$ is a tour with basis B and suppose x^* is a non-tour LP solution with basis B', computed from $\bar{x}$ by a non-degenerate pivot. Thus, there exists a primal-feasible basis B_0 for $\bar{x}$ such that B_0 differs from B' by a single primal simplex pivot. If the labels are getting a bit unwieldy, Figure 3.5 attempts to provide a blackboard-style schematic of the situation at hand. The solid arrow indicates the actual non-degenerate pivot computed, with the dashed arrow from B_0 to B' indicating the existence of some single pivot variable by which B_0 and B' differ. The double-ended, dotted arrow between B_0 and B just indicates that both correspond to the solution vector $\bar{x}$.

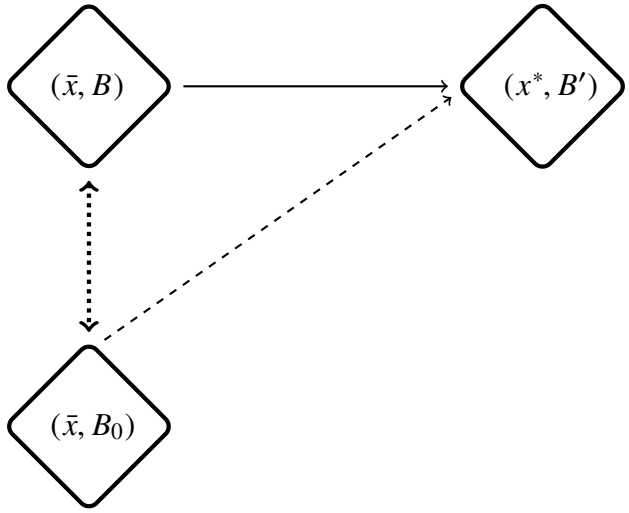


Figure 3.5: A non-degenerate pivot from $(\bar{x}, B)$ to (x^*, B') with basis B_0 for $\bar{x}$ that differs from B' by a single pivot.

Several matters are left unspecified by the definition of a non-degenerate pivot:

- Is B_0 distinct from B?
- If so, does the non-degenerate pivot operation give access to B_0?
- And if so, should we pivot back to B or to B_0?

Another matter is perhaps less obvious: how do we determine that x^* is distinct from $\bar{x}$? That is, what does it mean to say $\bar{x} \neq x^*$? We now enter a brief digression on tolerances, and on tradeoffs between practical efficiency and theoretical purity.

Most (but not all) linear programming codes perform their computations in floating point arithmetic, which is subject to various manners of roundoff error. It is typical to introduce a small positive 'zero' constant ε such that two floating point numbers α, β are considered equal if and only if $|\alpha - \beta| < \varepsilon$. Henceforth, $\alpha \neq \beta$ shall be understood to mean $|\alpha - \beta| \geqslant \varepsilon$. Similarly we define two floating point vectors x, y to be equal if and only if there is no index i such that $|x_i - y_i| \geqslant \varepsilon$, writing $x \neq y$ otherwise.

This suggests the possibility of differentiating between two vectors by heuristic means. To determine if two vectors x, y disagree, we must compare their entries one at a time. This means performing a number of arithmetic comparisons which is bounded below by the number of cities in the TSP instance. On the other hand, for a fixed cost vector c, it is also true that if $c^T x \neq c^T y$, then $x \neq y$. Certainly the same computational complexity is involved in obtaining the value $c^T x$, but this is usually computed as a matter of course during simplex computations, and it may be queried by OBJVAL.

It is true that an objective function may fail to separate distinct vectors. For example, Applegate et al. observe in Section 16.1 of [2] that the TSPLIB [28] instance u2319 has optimal tour length equal to the optimal objective value for the subtour polytope. Thus, the objective function provides a seminorm which sometimes erroneously assigns a distance of zero between distinct vectors. It is not clear how often this situation arises in practice, but theoretically it is of no real concern to "miss" a non-degenerate pivot in this manner, since vectors with identical objective values necessarily lie on the same face of the relaxation polytope.

In the remainder of this section, then, we shall speak of the abstract operation

DISTINCT(x, y) True if and only if x and y are regarded as distinct,

and this operation may be implemented exactly or heuristically as above.

3.3.2 Non-Degenerate Pivots via Pivot Limit

This non-degenerate pivot protocol requires the following operation to be supported by the LP solver:

PIV-LIMIT(n) Terminate PRIMAL-OPT after at most n pivots, whether or not optimality is certified.

This operation allows the non-degenerate pivot implementation outlined in Algorithm 3.3.1.

As discussed in Section 2.3.3, this is the approach taken by Letchford and Lodi in their work on primal cutting plane algorithms [18], as well as their Augment-Branch-Cut algorithms [19]. As we shall see in the forthcoming computational comparisons, and as reported by the authors themselves in their ABC research, Algorithm 3.3.1 performs quite poorly on degenerate linear programs (of which TSP relaxations are a particular example).

Algorithm 3.3.1: Non-degenerate primal pivots via iteration limit.

Data: A TSP tour $\bar{x}$ with associated basis B
Result: A non-degenerate primal pivot x^* with basis B'
Set $x^* \leftarrow \bar{x}$ and $B' \leftarrow B$
while x^* *is not dual feasible and* DISTINCT$(\bar{x}, x^*)$ *is false* **do**
    Call PIV-LIMIT(1) and PRIMAL-OPT
    Set $x^* \leftarrow$ SOLUTION
Set $B' \leftarrow$ BASIS
Return x^* and B'

3.3.3 Non-Degenerate Pivots via Objective Limit

Next is a novel approach to non-degenerate pivot computation. It is based upon the following operation.

LOW-LIMIT(α) Terminate PRIMAL-OPT when OBJVAL $< \alpha$, whether or not optimality is certified.

An LP solver which does not provide LOW-LIMIT directly may provide it as a special case of the following decidedly more general operation.

SET-CALLBACK(f) Call f after each pivot in PRIMAL-OPT, terminating if indicated by f.

Indeed, the operation SET-CALLBACK is so general that it contains LOW-LIMIT and PIV-LIMIT as a special case. For example if β is a target objective function bound, LOW-LIMIT could be implemented by calling SET-CALLBACK with a function which calls OBJVAL and sends the termination signal if OBJVAL $< \beta$. In any case, the resulting implementation is outlined in Algorithm 3.3.2.

Algorithm 3.3.2: Non-degenerate primal pivots via objective lower bound.

Data: A TSP tour $\bar{x}$ with associated basis B
Result: A non-degenerate primal pivot x^* with basis B'
Define β to be the objective value of $\bar{x}$;
Call LOW-LIMIT(β) and PRIMAL-OPT;
Set $x^* \leftarrow$ SOLUTION and $B' \leftarrow$ BASIS;
Return x^* and B';

It may not be immediately clear why Algorithm 3.3.2 provides an implementation of non-degenerate primal pivoting, so let us take a closer look. Recall that the specification of LOW-LIMIT

is to be able to prematurely terminate a course of primal optimization whether or not optimality has been certified. But if PRIMAL-OPT completes without interruption, this means a dual feasible basis for $\bar{x}$ has been computed. This leaves only the case of identifying a distinct pivot vector; but here the explanation follows from our discussion of the operation DISTINCT from Section 3.3.1. Indeed, since the TSP and any of its associated LP relaxations are minimization problems, PRIMAL-OPT will compute a solution with objective value never greater than the tour $\bar{x}$. So if one of the simplex pivots in a course of PRIMAL-OPT effects a change in objective value, a new vector x^* has been discovered, and the objective lower limit computes DISTINCT$(\bar{x}, x^*)$ implicitly.

So the end outcome of Algorithms 3.3.2 and 3.3.1 is the same: both perform single primal simplex pivots until encountering a distinct solution vector, thereby computing a non-degenerate pivot. The key difference is that Algorithm 3.3.2 consists of a single call to PRIMAL-OPT, whereas 3.3.1 repeatedly calls PRIMAL-OPT anew.

Applegate et al. provide a computational perspective on primal simplex degeneracy in Section 13.2.1 of [2]. To give a high-level summary tailored to the topic at hand, anti-degeneracy measures work *during* the course of primal simplex optimization, monitoring progress thus far and redirecting the solution process when appropriate. If the optimization is terminated and restarted with each individual pivot, then, from the point of view of the implementation, there is no progress to measure. This is the advantage of implementing non-degenerate pivots via objective limit rather than iteration limit.

3.3.4 Bases and Pivoting Back

We now return to the question of pivoting back to a tour after computing a non-degenerate pivot to a non-tour solution. Let us consider a pivot from a tour and basis $(\bar{x}, B)$ to a distinct LP solution and basis (x^*, B'). Recall that a non-degenerate pivot from $\bar{x}$ to x^* might consist of a sequences of primal-feasible bases

$$B, B_1, B_2, \ldots, B_j = B',$$

where for each i from $1, \ldots, j-1$, B_i is a basis for the original tour $\bar{x}$. When pivoting back to $\bar{x}$, the obvious question is whether we ought to pivot back to the basis B or the basis B_{j-1}. The less obvious question is, do we even have access to B_{j-1}?

Consider again the iteration limit implementation spelled out in Algorithm 3.3.1. This implementation is readily modified to obtain the intermediate bases $B_1, \ldots, B_{j-1}$. Note that the **while** loop of Algorithm 3.3.1 only executes if the solution vector is unchanged. Thus, if the loop body executes, we have moved from basis B_i to B_{i+1}, both of which are bases for $\bar{x}$. Thus, the calls to PIV-LIMIT and PRIMAL-OPT could be preceded by a call to BASIS to get the most recent basis for $\bar{x}$. The **while** loop fails to execute once we have pivoted to a new solution (x^*, B'), at

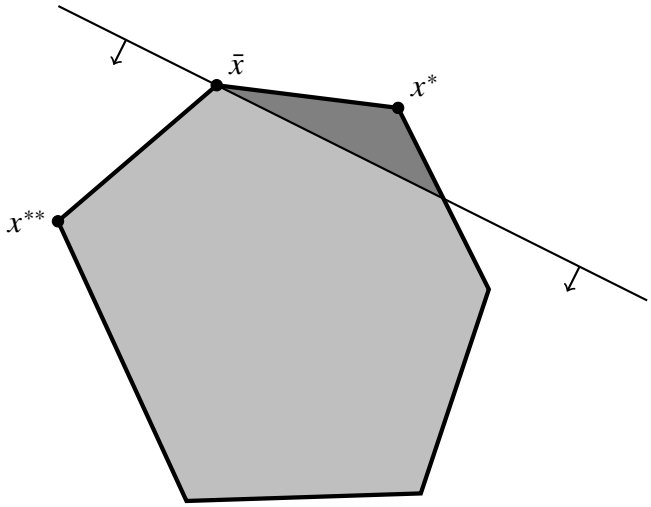


Figure 3.6: A non-degenerate pivot from $\bar{x}$ to x^* is cut off by a primal cutting plane, with x^{**} as a probable subsequent pivot.

which point the last basis stored is therefore a basis for $\bar{x}$ that differs from B' by a single primal simplex pivot.

Algorithm 3.3.2 can be adapted similarly. Explicitly, suppose we are using Low-Limit to implement objective value pivoting, and suppose moreover that Low-Limit is being implemented by a callback function f instated by Set-Callback. Thus, after each primal simplex iteration, the function f queries Objval to see if the objective value is less than the length of the best tour, in which case optimization is terminated. Consider a callback function f' which performs the same Objval check as f. But instead, if the objective value is unchanged, the callback f' could call Basis, again storing the most recent basis for $\bar{x}$.

Thus we have shown how, in either implementation, it would be possible to make a minor adjustment to get access to B_{j-1}. This brings us to the last implementation question, and possibly the least obvious. If we have access to B_{j-1}, do we in fact want to pivot back to it?

In Section 2.3.1 we discussed the impact of primal cutting planes, and the nature of primal cutting plane computation. Recall that primal cutting planes chop off line segments between tours and non-degenerate pivots, redirecting the optimization at each step. Figure 3.6 sketches this situation, contrasting with the standard case in Figure 3.7.

In Section 13.1.3 of [2], Applegate et al. describe the dual (steepest edge) simplex algorithm as "the algorithm of choice in almost all of [their] TSP computations." The whole of Chapter 13 of [2] provides a more thorough grounding in this choice, but the idea is simple: dual fractional algorithms move from one dual lower bound to the next, and the basis for the previous optimal solution gives a starting basis for the next optimization. Indeed, robustness of bases

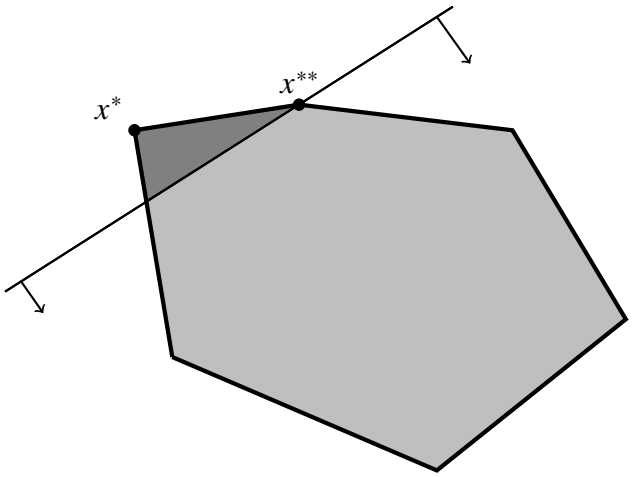


Figure 3.7: An optimal LP solution x^* is cut off by a standard cutting plane, with x^{**} as a probable subsequent optimal LP solution.

under the addition of constraints is a strong reason to favour the dual simplex algorithm for such computations. Thus in Figure 3.7 we mean to suggest that the vectors x^* and x^{**} may have quite similar bases, and that we can get from one to the other with relative ease. Figure 3.6 illustrates precisely the opposite of this phenomenon: we have no reason to suspect that bases for successive pivots x^* and x^{**} will be similar; and in fact we actively expect this *not* to be the case.

At this point we run into a limitation of attempting to develop intuition using two-dimensional diagrams. It is unlikely that either of these Mickey Mouse figures give an accurate picture of what primal TSP degeneracy looks like. In the primal case, it is intuitively appealing to reason that the last primal feasible basis for $\bar{x}$ was geometrically close the basis for x^*. But given that we cut off the entire line segment going in this direction, will such a basis be helpful for getting to a new pivot such as x^{**}? Camargue uses the implementation of Algorithm 3.3.2, without any modification for basis copying. The choice of Algorithm 3.3.2 as written is justified thoroughly in the next section, but I do not claim to have made a rigorous study of the basis copying step. This decision is motivated entirely by the facilities provided by the LP solver used to implement Camargue.

At any rate, we have raised more than a fair number of theoretical conundrums at this point. The hope is that these discussions and definitions have complicated our understanding of what was regarded as a more simple matter in the extant literature. In the next section, we seek computational answers to some of the questions raised, obtaining results that will be used to justify implementation choices in Camargue.

3.4 Computation

The question we try to answer is the most general one, and the one with the most decisive answer. How should non-degenerate pivots be computed? The hypothesis is that non-degenerate pivot computation should be regarded as truncated primal optimization, because pivoting by iteration limit may destroy the ability of the LP solver to monitor progress and apply anti-stalling measures. To make this precise, Table 3.1, compares the performance of non-degenerate pivot implementations at optimizing LP relaxations.

For all TSPLIB [28] instances of size between 3,000 and 15,000 nodes, indicated in the "Instance" column, we consider an initial edge set consisting of the union of 10 Lin-Kernighan tours, as implemented by Concorde [2]. Then, the degree LP for this edge set is loaded into the IBM ILOG CPLEX 12.6 LP solver. In all examples, the devex pivot selection rule is chosen as the primal simplex pivot algorithm, and the starting basis is that computed by the Padberg-Hong [25] approach discussed in Section 3.1.

Within the sub-table for each instance, the column "Protocol" indicates the means by which the degree LP is optimized. For rows labelled Primal-Opt we just use the CPLEX implementation of primal simplex optimization. In rows labelled Low-Limit, we adapt the non-degenerate pivot implementation of Algorithm 3.3.2 to the task of primal optimization: Algorithm 3.3.2 is called repeatedly, each time querying the objective value of the pivot obtained and then setting this as the new low limit until reaching optimality. Finally, for rows labelled Piv-Limit Algorithm 3.3.1 is called repeatedly until the solution is dual feasible.

The column "CPU Time (scaled)" reports CPU times for each protocol as a ratio over the time taken for Primal-Opt; it seems reasonable to use actual optimization as the standard to judge by. The CPU times themselves are geometric means of five separate trials in which the edge set and degree LP are loaded anew, and then the respective protocol is used to optimize the degree LP. Each trial is conducted with an identical random seed used for generating the initial edge set.

Finally, the column "Iteration Count" reports the number of primal simplex pivots performed before reaching optimality. Since all of the trials have identical starting edge sets (and therefore LP relaxations), the number of pivots is the same each time.

As we would expect, Table 3.1 shows that actual optimization is the fastest way to optimize an LP by a wide margin. After that, though, Low-Limit is anywhere from four to ten times faster than Piv-Limit.

It is interesting to note that actual iteration counts do not appear to correlate with solution speed. In all cases, Primal-Opt performed the greatest number of iterations while always attaining the fastest speed. On the other hand, Low-Limit and Piv-Limit often perform a comparable number of iterations, but Low-Limit is always a great deal faster.

Table 3.1: Comparison of pivot protocols as optimizers.

Instance	Protocol	CPU Time (scaled)	Iteration Count
pcb3038	PRIMAL-OPT	1.00	1357
	LOW-LIMIT	11.56	725
	PIV-LIMIT	59.56	780
fl3795	PRIMAL-OPT	1.00	805
	LOW-LIMIT	4.62	604
	PIV-LIMIT	45.52	632
fnl4461	PRIMAL-OPT	1.00	2417
	LOW-LIMIT	14.49	1050
	PIV-LIMIT	77.19	1186
rl5915	PRIMAL-OPT	1.00	1360
	LOW-LIMIT	12.77	607
	PIV-LIMIT	67.79	744
rl5934	PRIMAL-OPT	1.00	1425
	LOW-LIMIT	9.57	552
	PIV-LIMIT	50.59	758
pla7397	PRIMAL-OPT	1.00	3200
	LOW-LIMIT	7.71	1447
	PIV-LIMIT	82.04	1844
rl11849	PRIMAL-OPT	1.00	4026
	LOW-LIMIT	23.53	1347
	PIV-LIMIT	123.75	1926
usa13509	PRIMAL-OPT	1.00	6963
	LOW-LIMIT	30.16	2664
	PIV-LIMIT	147.47	3328
brd14051	PRIMAL-OPT	1.00	7998
	LOW-LIMIT	30.80	2234
	PIV-LIMIT	166.72	3732

The tests in Table 3.1 are meant to computationally back the speculations about anti-stalling measures taken by CPLEX. Of course, optimizing the degree LP is not at all indicative of how Algorithms 3.3.1 or 3.3.2 would actually be used during primal cutting plane TSP computation.

Table 3.2 shows the results with tests on a slightly expanded test bed, consisting of all TSPLIB instance of size between 2,000 and 15,000 nodes. (The instances in the 2,000 node range were omitted from Table 3.1 purely for formatting purposes.) The initial edge sets are the same, as

are the initial bases. Tests were run on a 3.5 GHz Intel Xeon server, with the columns "CPU Seconds" indicating a geometric mean of five independent trials as described for Table 3.1. This time, we compare the performance of the two implementations on two LP relaxations as follows. We compute a non-degenerate pivot from the degree LP, and then call separation routines for segment cuts, connect cuts, fast blossoms, and block combs; a garden variety choice of separation routines. All cuts found are added to the LP, at which point we pivot back to the tour and compute a new non-degenerate pivot.

For each TSPLIB instance, we have the column groups "Degree Pivot", for pivoting from the tour with the degree LP active; and "Cuts Pivot" for pivoting from the tour once cuts have been added. The row Low-Limit indicates the implementation of Algorithm 3.3.2, and the row Piv-Limit indicates the implementation of Algorithm 3.3.1. Moreover, we include the column "Value" to indicate the objective value of the pivot computed.

The column of objective values reveals an interesting subtlety which is not indicated by the results in Table 3.1. Namely, the non-degenerate pivot implementation affects not only the speed of pivot computation, but also the direction as well. Only in the case of u2319 with the degree LP do the Low-Limit and Piv-Limit pivots agree in objective value. In all other trials, the two implementations compute distinct pivots.

As for computation speed, it seems clear that the Low-Limit implementation is faster, but the relative speedups may be hard to grasp given that almost all computations completed in under a second of CPU time. Figure 3.8 attempts to illustrate the speedups obtained. The vertical axis scale, "Speedup factor", is obtained by taking the CPU time ratio of Piv-Limit to Low-Limit for the times reported in Table 3.2. Thus, the bars show how many times faster the Low-Limit implementation is.

With the results in Table 3.2, and the illustration in Figure 3.8, we now conclude decisively that implementation with Low-Limit is the most computationally effective. The "Degree Pivot" for brd14051 is the only case attaining a speedup factor of less than two. In all other trials, the Low-Limit implementation is at least four times faster, with some more exorbitant speedups in the range of 20-30 times for the "Cuts Pivot" on rl5934, usa13509, and brd14051. The fact that the most dramatic speedups are attained with cuts present is no accident: it appears that the strength of the Low-Limit implementation is that it continues to quickly compute non-degenerate pivots even as a large number of cuts are added. Extremely early implementations of Camargue computed non-degenerate pivots using Algorithm 3.3.1 through the entirety of the primal cutting plane solution process, and the speedups obtained upon discovering the approach of Algorithm 3.3.2 were comparable to, and often in excess of, the most dramatic speedups reported here.

Table 3.2: Comparison of non-degenerate pivot implementations.

Instance		Degree Pivot		Cuts Pivot	
		CPU Seconds	Value	CPU Seconds	Value
d2103	Low-Limit	0.001643	81469.50	0.006786	81831.00
	Piv-Limit	0.031598	79285.50	0.040469	81469.50
u2152	Low-Limit	0.003178	65267.00	0.003000	65317.00
	Piv-Limit	0.031989	65229.00	0.048782	65236.50
u2319	Low-Limit	0.003287	234970.00	0.007583	234929.00
	Piv-Limit	0.068587	234970.00	0.056191	234970.00
pr2392	Low-Limit	0.004182	382278.50	0.006787	382290.00
	Piv-Limit	0.069989	382290.00	0.072387	382120.00
pcb3038	Low-Limit	0.007999	139317.00	0.004999	139266.00
	Piv-Limit	0.080187	139275.00	0.092785	139229.50
fl3795	Low-Limit	0.002999	28459.00	0.007788	28791.00
	Piv-Limit	0.019391	28460.50	0.034392	28460.50
fnl4461	Low-Limit	0.014178	184195.00	0.025793	184191.00
	Piv-Limit	0.242763	184115.50	0.260960	184095.00
rl5915	Low-Limit	0.007788	577980.50	0.016390	579512.50
	Piv-Limit	0.053791	579559.00	0.089185	580001.50
rl5934	Low-Limit	0.015390	563099.00	0.003365	562982.00
	Piv-Limit	0.107582	562879.00	0.133578	562995.00
pla7397	Low-Limit	0.013792	23493912.00	0.025391	23492943.00
	Piv-Limit	0.200368	23464888.00	0.269759	23468649.00
rl11849	Low-Limit	0.035995	939953.50	0.043791	943975.00
	Piv-Limit	0.301553	939497.50	0.388939	940108.00
usa13509	Low-Limit	0.087383	20239725.00	0.035591	20235296.00
	Piv-Limit	1.539163	20233773.00	1.192419	20234002.50
brd14051	Low-Limit	0.050586	472822.50	0.034591	475324.50
	Piv-Limit	0.083586	472837.50	1.031840	475308.50

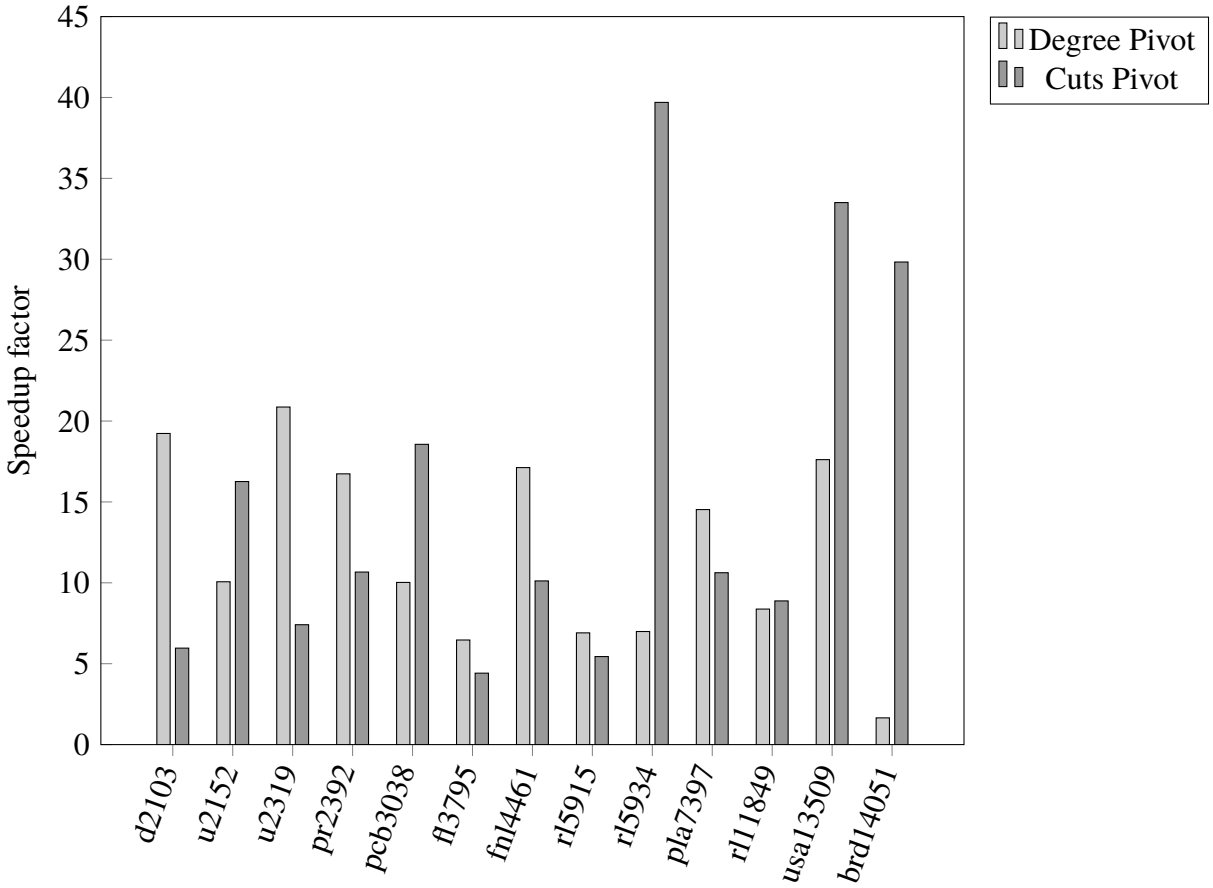


Figure 3.8: Visualization of speedups from Table 3.2.

3.5 Concluding Remarks and Further Reading

All our discussions in this chapter, implicitly or otherwise, have been underlain by a fixed choice of pivot selection rule. That is, when speaking of bases that differ from each other by a single primal simplex pivot, and contrasting this with non-degenerate pivots, such measures of "single" or "many" pivots being needed are defined relative to a *pricing* rule which chooses a pair of entering and leaving variables. Common choices include full pricing, multiple pricing, the devex rule, and the steepest edge criterion. Applegate et al. give a more detailed discussion in section 13.5 of [2].

Among primal pricing rules, the main tension is a trade-off between computational effort and efficacy on degenerate linear programming problems. Recall for example that in Section

2.3.2, we reviewed the algorithmic approach taken by Padberg and Hong [25], who chose to use the steepest edge selection rule. The steepest edge rule is quite computationally intensive, but it performs better on average with highly degenerate linear programs. Even if we choose a pricing rule such as devex or steepest edge that performs better on degenerate LPs, we still expect that many degenerate single pivots will be encountered on the way to identifying a proper non-degenerate pivot. Indeed, when mentioning the pivot sequence from $(\bar{x}, B)$ to (x^*, B'), we mentioned the existence of $(\bar{x}, B_0)$ which differs from B' by a single primal simplex pivot. (This was the situation sketched in Figure 3.5.) But we also asked whether it must necessarily be the case that B_0 and B are distinct.

Some relatively new research regarding primal pricing rules suggests that perhaps B and B_0 need not be distinct. In 2011, Raymond, Soumis, Metrane, Towhidi, and Desrosiers [27] published a paper describing a pricing rule referred to as the *positive edge criterion*. They prove that, depending on the floating point arithmetic model, the probability of identifying a degenerate pivot is 2^{-30} or 2^{-62}. They also report computationally that this selection rule shows great potential in practical applications on highly degenerate linear programs. Towhidi, Desrosiers, and Soumis [29] report on a computational study where the positive edge criterion was implemented directly in the COIN-OR CLP linear programming solver, also obtaining promising results. In both papers, however, the positive edge criterion is used as a pricing protocol for primal simplex optimization. Thus, from the point of view of primal cutting plane computation, it would be interesting to test the efficacy of the positive edge rule simply as a protocol for implementing non-degenerate pivots.

Chapter 4

Primal Separation Algorithms

What cuts do we want to use, and how are we going to find them? The most widely employed cuts in the solution of the TSP conform to the *template paradigm*. For example, subtour elimination constraints are a template of cuts. Other templates we will consider are blossoms and combs, and domino parity (DP) inequalities. The class of DP inequalities is an order of magnitude more complex than any of the former classes, containing each of them as a particular special case. These are not discussed in this chapter; rather the entirety of the next chapter is devoted to a particular subclass of DP inequalities, the so-called *simple* DP inequalities.

Suppose we have a TSP instance specified by a graph $G = (V, E)$, and a basic feasible solution x^* to some LP relaxation of the TSP polytope. Then we can define the *support graph* associated to x^*. It is the graph G^* with vertices V and edges $\{e \in E : x^*_e > 0\}$. Heuristic and exact separation routines generally involve computations on the graph G^* with edge weights derived from x^*, or graphs further derived from G^*.

In the dual fractional approach, it is often the case that a mix of heuristic and exact separation routines are employed, even when polynomial time exact separation algorithms are known. Indeed, a polynomial running time may still not be efficient enough for an algorithm that will be called hundreds or even thousands of times over the course of the solution process. As pointed out by Letchford and Lodi [20] however, primal separation for various TSP cuts tends to be conceptually simpler than standard separation, and with some cuts there is a polynomial factor speedup.

Another type of cuts we shall have occasion to consider are the *general purpose* cuts, namely, Gomory mixed integer cuts. As the name suggests, these are applicable (and indeed fruitfully employed) in the solution of general mixed integer programs, and they operate with no respect for any underlying combinatorial structure.

The rest of this chapter proceeds as follows. We will establish heuristic and exact separation

routines for certain primal separation problems, making comparisons between available standard separation algorithms in terms of what is suitable for practical use. We emphasize at all times the interplay between primal and standard separation, making reference to computational studies which employ a mix of standard and primal separation algorithms, despite opting for one or the other approach with regards to the linear programming problems. This is the very situation adopted in this thesis, and implemented in Camargue: cut separation is driven by primal separation algorithms, but we also make liberal use of various standard heuristics for separating subtours and blossoms. These will be detailed in due course over the next two sections, with additional remarks in Section 4.4.

Discussions in this chapter will attempt to provide survey reviews of algorithms elucidated in other papers, while providing more analysis on any practical improvements which are regarded as original research in this thesis.

4.1 Subtour Elimination Constraints

Subtour inequalities are the oldest and most well-understood template of cuts for the TSP. They take the form

$$x(\delta(S)) \geqslant 2 \text{ for all } S \subset V \text{ such that } 2 \leqslant |S| \leqslant |V| - 2,$$

from which it is clear that they correspond precisely to cuts of weight less than two in G^*, with edge weights given by x^*. Indeed, this observation leads to the well-known polynomial-time exact separation algorithm for subtour inequalities, in which SECs can be found by computing $|V| - 1$ minimum cuts on G^*.

Although this algorithm may run with pleasant efficiency, Applegate et al. (See Chapter 6 of [2]) take the view that it should be used as something of a mild last resort. For example, if x^* induces a disconnected support graph G^*, then each connected component of G^* gives a subtour cut violated by x^*. The highly efficient *connected component* heuristic searches for SECs of this type, essentially just performing a depth-first search of the support graph.

Applegate et al. also employ a heuristic for generating what they refer to as *segment cuts*. These are SECs where the vertex set is simply a set of nodes which appear consecutively in some tour $\bar{x}$. Presently, we shall see that segment cuts are in fact primal subtour cuts. That is, as observed by Letchford and Lodi [20], and detailed by Applegate et al. in Chapter 6 Section 3 of [2], primal subtour separation can be used as a fast heuristic for standard subtour separation.

4.1.1 Exact Primal Separation of Subtour Inequalities

Recall that, given a tour $\bar{x}$ and an LP solution x^*, the primal separation problem for $\bar{x}$ and x^* is to find a cut violated by x^* which holds at equality at $\bar{x}$. Thus, a nonempty proper subset S of V solves the primal separation problem for subtour inequalities if $\bar{x}(\delta(S)) = 2$ but $x^*(\delta(S)) < 2$. If $|V| = n$, let us suppose for notational convenience that the indices $1, \ldots, n$ have been permuted so that the tour $\bar{x}$ is given by

$$(1, 2, \ldots, n).$$

Then primal subtour cuts correspond precisely to subsequences

$$(i, \ldots, j) \text{ where } j \in \{i + 1, \ldots, i + n - 2\},$$

with indices are taken modulo n. Thus, we call them segment cuts because they correspond precisely to contiguous tour segments. To get a visual, Figure 4.1 illustrates a violated primal subtour inequality on a six-node TSP instance, a classic blackboard example. Figure 4.1a shows

(a) A tour with a highlighted tour segment.

(b) An integral subtour.

Figure 4.1: A violated primal subtour inequality.

the edge set of a tour vector $\bar{x}$, and Figure 4.1b shows an integral subtour vector x^* which violates a primal SEC with respect to $\bar{x}$. If S denotes the set of vertices incident with the grey-shaded edges in Figure 4.1a, it is clear that $\bar{x}(\delta(S)) = 2$, but $x^*(\delta(S)) = 0$. Moreover, the grey edges are a contiguous segment of the tour edges in $\bar{x}$.

It is easy to see that the number of these segments, and hence the number of candidate sets for a primal subtour cut, is $n(n-3)/2$. In the standard case, the number of candidate sets for a subtour cut is on the order of 2^n. This is enough to hint that finding primal SECs should be easier, which is indeed the case. Padberg and Hong [25] devise a primal separation algorithm for SECs that runs in time $O(mn)$, reporting that their procedure "determines the *most violated* subtour elimination constraint if any exists". A slightly closer reading of the details of their algorithm shows that it is readily modified (with no change in running time) to return a collection of violated subtour constraints, rather than just the most violated one.

Remarking on the Padberg-Hong approach, Letchford and Lodi [20] report that with "a little more work," a runtime of $O(m \log n)$ is possible. This is the approach taken by Applegate et al.,

who reduce the problem to finding the minimum element from a heap of prefix sums. In fact, Letchford and Lodi observe the following.

> We have since found out that this idea was discovered independently by Applegate et al. (1998) a number of years ago . . . The interesting thing as that (as we understand it) Applegate et al. (1998) did not have primal separation in mind. Rather, they ran this algorithm, using the best tour $\bar{x}$, as a fast heuristic for *standard* separation.

Like the Padberg-Hong approach, the implementation of Applegate et al. easily returns a good sample of violated segment cuts, if any exist. To explain in a bit more detail, we adopt the notation used by Applegate et al. in Section 6.3 of [2]. If $1 \leqslant i \leqslant t \leqslant n-1$, let I_{it} denote the segment $\{i, i+1, \ldots, t\}$. For each i from $1, \ldots, n-1$, their algorithm returns the interval I_{it} minimizing $x^*(\delta(I_{it}))$ over all t greater than i. This approach strikes a compromise between finding a maximally violated cut and finding every single violated cut. But as regards the task of finding every single violated cut, Applegate et al. describe an efficient approach for doing this, too. We shall examine this procedure in more detail in the following chapter, this time in the context of separating simple domino parity inequalities.

4.1.2 Primal-Inseparable Connected Component Cuts

There is one matter left untreated in the preceeding discussion of primal SECs. Namely, if we pivot from some tour $\bar{x}$ to a new LP solution x^*, then x^* may be an integral subtour vector which admits no SECs arising from segments. We illustrate this situation in Figure 4.2 with a simple TSP instance on 14 nodes. Figure 4.2a shows a tour on this instance, and Figure 4.2b shows an integral subtour vector which admits no primal subtour cuts. To see why this is the case, we examine the figures a bit more closely. Let $\bar{x}$ denote the tour in Figure 4.2a, and let $\hat{x}$ denote the

(a) A tour. (b) An integral subtour.

Figure 4.2: A tour together with an integral subtour vector which admits no primal subtour cuts.

integral subtour vector in Figure 4.2b. It is clear that $\hat{x}$ sports a violated connected component SEC, but it may be less clear why this is not a primal SEC. Recall that if a vertex subset S is to give a primal SEC for $\hat{x}$, then S must correspond to a contiguous segment of the tour $\bar{x}$. Examining

Figure 4.2, the connected components of $\hat{x}$ give a violated SEC, but the vertex sets of these connected components do *not* coincide with contiguous subsequences of the tour defining $\bar{x}$. For an alternative explanation, recall also that if S is a connected component of $\hat{x}$, then there must hold $\bar{x}(\delta(S)) = 2$ for S to give a primal SEC. Figure 4.3 shows clearly that, for either equivalent connected component in Figure 4.2b, we have $\bar{x}(\delta(S)) = 4$.

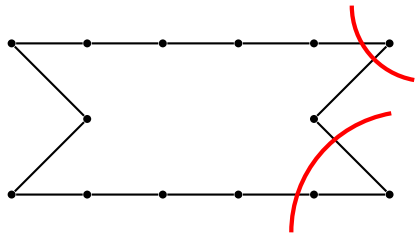


Figure 4.3: The edge set of the primal inseparable cut from Figure 4.2.

To be clear, our other primal separation routines may also fail to return primal cuts when standard cutting planes exist. The concern here is that an integral non-tour solution vector also leaves us with no fractional variables with which to commence a branch-and-cut search. Thus, we need some way of handling this situation.

Letchford and Lodi [18] describe a possible approach.

> If x^* is integral but not feasible, find an inequality . . . which is violated by x^* yet *not tight* at $\bar{x}$, add it to the LP and perform a single (dual) simplex pivot to arrive at the new (fractional) point $\hat{x}$ which is a convex combination of $\bar{x}$ and x^*. Set $x^* := \hat{x}$.

Then, they say to use a fractional structural variable from $\hat{x}$ to generate a Gomory cut which is tight at $\bar{x}$ and cuts off $\hat{x}$, hence cutting off x^* since x^* lies on the line segment through $\bar{x}$ and $\hat{x}$. At this point, they pivot back to $\bar{x}$ and delete the slack connected component cut, which is no longer needed.

The main issue with this procedure is the ready availability of tight Gomory cuts. Given a solution x^* with fractional x^*_i, it is a matter of elementary integer programming theory that x^* violates a Gomory mixed-integer cut generated from the tableau row associated to x^*_i. But the numerical safety of such a cut is another matter entirely. The entirety of Section 4.3 is dedicated to the topic of safely generating primal Gomory cuts, so for now suffice it to say that the existence of such cuts cannot be taken for granted.

In their study of the degree-constrained minimum spanning tree (DCMST) problem, Behle et al. encounter a similar difficulty. Indeed, IP solution of the DCMST requires adding cutting planes for so-called cycle elimination constraints, which are an analogue of TSP subtour cuts.

Thus, Behle et al. also have to handle the situation of an integral solution vector which is not a feasible solution. They write that, "for esthetical reasons", they do not ever invoke the dual simplex algorithm in their approach. Instead, they simply add the non-tight cut, using the dummy LP procedure discussed in Section 3.1 to get a new primal feasible basis for the current solution.

In Camargue, I adopt an approach more in the spirit of Behle et al. Suppose x^* is obtained from $\bar{x}$ by non-degenerate pivot, but x^* admits no primal SECs. Then pivot back to $\bar{x}$, add the component cut, and pivot to a new x^*; repeat this until x^* is connected. In the case that x^* is a new tour, augment, setting $\bar{x}$ equal to x^*. If not, call primal separation for other known classes of inequalities.

Of course, it still may happen that a round of connected component cuts results in a fractional solution for which no other primal cuts can be found. In this case, we at least have a collection of candidate edges that can be used to initiate a branch-and-cut search, if desired.

4.2 Blossom Inequalities

The next cut template we shall consider is that of blossom inequalities. Blossoms are defined by a nonempty subset H of V, the handle, and an odd-cardinality subset of edges F contained in $\delta(H)$, the teeth. Among other possibilities, they can be written as

$$x(\delta(H) \setminus F) + \sum_{e \in F} (1 - x_e) \geqslant 1. \tag{4.1}$$

In 1982 Padberg and Rao released the landmark paper [26] which gives a polynomial time exact standard separation algorithm for blossom cuts. This result came a few years too late for Padberg and Hong's 1980 paper [25] in which the authors resort to heuristic separation of blossom inequalities.

Although polynomial, the Padberg-Rao algorithm is not necessarily suitable for practical computation. It involves computing $O(m)$ max flows on a graph with $O(m)$ vertices and $O(m)$ edges. And Applegate et al. report in Chapter 16 of [2] that exact blossom separation is not used by default in their Concorde solver.

In this section we treat the question of how to separate blossom inequalities in the primal case. Consider the support graph G^* induced by a non-tour pivot vector. In their primal cutting plane TSP study [25], Padberg and Hong describe a fast blossom heuristic which examines odd components of the graph induced by the fractional edges of G^*. As we discussed in Section 2.3.2, this is essentially a standard heuristic with an added step of checking tightness at the current tour; Applegate et al. describe using the same heuristic in Section 7.1 of [2]. In the same section,

Applegate et al. describe a slightly more involved heuristic due to Grötschel and Holland, based on similar principles. In Camargue, the approach to blossom separation is first to try and find blossoms using one or the other of these fast standard heuristics.

If the fast heuristics return no violated blossoms, we turn to the exact separation algorithm described by Letchford and Lodi [20]. Letchford and Lodi's algorithm requires $O(m)$ max flows to be computed directly on the support graph G^*, giving a polynomial-factor speedup over the Padberg-Rao algorithm. The next section is concerned specifically with an implementation of this exact separation algorithm, and in particular with an approach for running the minimum cut computations concurrently.

With this discussion, and the implementation described in the next section, I make the claim that for practical purposes, the complexity of primal blossom separation is comparable to that of standard SEC generation. By this I mean that in Camargue, blossoms are separated either by one of two fast heuristics or by an efficient exact separation algorithm. And in the standard case, this is precisely the state of affairs with SEC generation in, for example, the Concorde code of Applegate et al. [2].

4.2.1 Exact Separation

In this section, we study the exact primal blossom separation algorithm due to Letchford and Lodi [20]. This algorithm is asymptotically faster and conceptually simpler than its standard counterpart: the Padberg-Rao algorithm [26] requires a series of $O(m)$ maximum flow problems on a graph with $O(m)$ vertices and $O(m)$ edges. Letchford and Lodi [20] also observe that these maximum flows

> are not 'independent' of each other, in the sense that the result of each maximum flow computation influences the choice of source and sink nodes in the subsequent one.

Letchford and Lodi's primal separation algorithm still requires $O(m)$ max flows, but these are computed directly on the support graph G^*, which of course has n vertices and m edges. Moreover, in Letchford and Lodi's algorithm "each maximum flow computation can be carried out independently". This seems clearly to allude to the possibility of parallelizing the separation algorithm, which is the approach adopted in Camargue. To explain, we consider a presentation of their algorithm which differs only superficially from that given in [20].

The idea is to define a series of modified edge capacities on the support graph $G^* = (V, E^*)$. First partition E^* into the subsets

– $E^0 = \{e \in E^* : \bar{x}_e = 0\}$, the set of support graph edges which are not in the tour; and

– $E^1 = \{e \in E^* : \bar{x}_e = 1\}$, the set of support graph edges which are in the tour.

Then define a weight w on G^* by $w_e = x^*_e$ for each e in E^0 and $1 - x^*_e$ for each e in E^1. For each edge e, we then define a modified weight function w' which agrees with w except that w'_e is $1 - x^*_e$ for each e in E^0 and x^*_e for each e in E^1. With u, v as the endpoints of e, we then compute a minimum u, v-cut for each such E. If the cut has weight less than one, let H be the shore of the cut found. Then H gives the handle of a violated blossom inequality, with teeth $\{e\} \cup (\delta(H) \cap E^1)$ if $e \in E^0$, and $(\delta(H) \cap E^1) \setminus \{e\}$ if $e \in E^1$. At this point we revert to the weight function w and proceed to the next edge in the support graph.

Evidently the bottleneck of this algorithm is the $O(m)$ minimum cuts, but these computations may be run concurrently in exchange for an overhead in memory. Indeed, to obtain w' from w, one might either change the corresponding entry of w, or create a copy of w with the appropriate change. It is reasonable to assume that a minimum cut algorithm requires access to the edge capacity function as input, hence creating w' as a copy enforces mutual exclusion on the minimum cut computations.

4.3 Safe Gomory Cuts

The final topic of this chapter is that of Gomory cuts. Unlike the other cuts considered in this thesis, Gomory cuts do not arise from a template specific to the TSP, and they do not make use of any combinatorial structure of the TSP, or an LP relaxation support graph. Rather, they are a particular instance of so-called *mixed integer rounding (MIR)* cuts, which are generated by aggregating and rounding valid inequalities for a given LP relaxation. In the case of Gomory cuts, the valid inequalities used are rows of the simplex tableau for a fractional LP solution.

In [18], Letchford and Lodi include the generation of Gomory cuts as a possible step in their basic primal cutting plane framework. The algorithm presented by Letchford and Lodi is deliberately extremely general, but the basic idea is that it should be tailored to solving a particular class of MIP problem for which we can generate "strong", hopefully facet-defining primal inequalities. Thus, pivoting from a resident best solution $\bar{x}$ to a vector x^*, we first try to find strong primal cutting planes for $\bar{x}$ and x^*. If the primal algorithms return no strong inequalities, we resort to generating general-purpose cuts, such as the Gomory cuts. The authors provide a helpful sketch of how we can generate primal Gomory cuts.

> Consider a mixed-integer cut generated from the row of the simplex tableau associated with variable x_i, where x^*_i is fractional. It is well-known that this cut is a strengthened version of a so-called intersection cut derived from the disjunction $(x_i \leqslant 0) \vee (x_i \geqslant 1)$. The intersection cut is by definition tight at all n points which can be obtained by

> increasing the value of a non-basic variable until x_i becomes equal to either 0 or 1. One of these points is $\bar{x}$ itself. Therefore the intersection cut is tight at $\bar{x}$. The mixed-integer cut, being at least as strong, is therefore also tight at $\bar{x}$.

Thus, we are equipped with a theoretical guarantee on the existence of a violated Gomory cut that solves the primal separation problem. But in Section 4.1.2 we alluded to some difficulties that may arise regarding the safety and numerical stability of the cuts generated.

In their paper on primal branching and cutting [19], Letchford and Lodi address this concern specifically when describing the conditions under which branching is invoked after cutting plane generation: "After 25 consecutive rounds of Gomory fractional cuts, in order to avoid numerical problems, we branch."

Behle et al. [3] apply Letchford and Lodi's primal cutting plane framework to the degree-constrained minimum spanning tree problem. If no combinatorial cuts are found, the authors attempt to

> separate a fractional point x^* by a Chvátal-Gomory cut that is tight at $\bar{x}$ and only contains integral coefficients. So at all times our matrix A is integral.

Thus Behle et al. take a very different approach from that of Letchford and Lodi, but numerical stability appears to have been a concern for them as well. At any rate, these two studies both show that one cannot "just" generate Gomory cuts: precautions are necessary to preserve numerical stability.

Indeed, the short story behind Gomory cuts is that they were discovered by Ralph Gomory in 1960 [11], and for the ensuing few decades they were widely considered computationally useless due to the numerical difficulties they posed. Thanks to some research advances of the 1990s, they are now regarded as a vital component of effective solvers for large scale MIP problems (see e.g., Bixby et al. [4]).

In TSP computation, the trajectory is almost the exact opposite. In Sections 4.3 and 4.5 of [2], Applegate et al. describe computational studies published from the 1960s to mid 1980s which considered Gomory cuts as a primary engine of TSP solution. In the time since then, the template paradigm assumed dominance, and Gomory cuts have been all but absent from TSP computation. The next section describes an interesting exception, together with the numerical safety framework employed for generating Gomory cuts in Camargue.

4.3.1 The Separation Framework

The approach considered in this thesis is the framework of Cook, Dash, Fukasawa, and Goycoolea [6] for generating numerically safe Gomory cuts in floating point arithmetic. The paper [6]

provides a more thorough account of the numerical difficulties associated with computing MIR cuts, as well as describing possible remedies adopted by other researchers. Thus, with [6] as a reference, we may explicitly name the spectre of numerical instability that has been alluded to. Namely,

- the cuts found may be very *dense*, assigning nonzero coefficients to a large portion of variables; and
- errors in floating point arithmetic may result in invalid cuts being generated, i.e., cuts which are violated by otherwise-feasible solutions.

I have chosen to adopt the Cook et al. approach because it provides *provable* (rather than heuristic) numerical safety in a straightforward way, and it does so using a floating point arithmetic model, making it easy to integrate into Camargue. Moreover, the authors provide their source code as a digital supplement to the publication of [6], making it an appealing choice from an implementation point of view. As regards the first issue, of cuts being dense and making LP relaxations harder to solve, the procedure of Cook et al. provides no claims of amelioration. In fact, they report that in their own implementation

> unsafe cuts are slightly sparser on average than the safe ones, which is somewhat surprising, considering that we are not taking any measures to achieve this goal.

As regards industry-standard implementations of MIR cuts, they remark that

> by using safe cuts, one cannot use some of the ad hoc measures that are usually applied to unsafe cuts to make them more numerically stable and/or more safe–for example, rounding the coefficients to zero when they are very small. Unsafe cuts that use these safeguards can have advantages over safe cuts, such as being sparser, so these issues have to be weighed more carefully in those cases.

Thus, we shall engage in a study of cut sparsity tailored directly to the primal perspective.

The algorithmic approach in Camargue is to apply the MIR procedure almost exactly as described in Section 5 of [6], tailored to the sketch given by Letchford and Lodi in [18]. Thus the change is that we only consider as candidates the fractional structural variables in the current basis, rather than considering basic slacks as well. If the combined approach of Letchford-Lodi and Cook et al. returns primal Gomory cuts, we are guaranteed that these cuts will never erroneously cut off a feasible solution. However, it is possible that the safe rounding and aggregation procedure of Cook et al. will fail to return violated Gomory cuts, even when the existence of such cuts ought to be guaranteed by the presence of one or more fractional basic variables. Thus, we do not

rely on Gomory cuts as a fail-safe to advance the solution process when no other primal cuts are found.

We pause to point out an interesting feature of the study of Cook et al. [6], which has consequences for the implementation choices made in Camargue. Cook et al. perform some computational experiments on TSPLIB [28] instances, using safe Gomory cuts to improve the LP relaxation lower bound given by the Concorde TSP solver [2]. As discussed by Applegate et al. in Chapter 12 of [2], and as we shall discuss in the primal context in Chapter 6, a key feature of any TSP solver is having some method for edge pricing which allows only a subset of edges to be considered at a time, all the while being able to prove that bounds or optimal solutions on partial edge sets are indeed valid for the full edge set of the instance. Moreover, if edges are added to the core LP, we must be able to recompute existing cuts where necessary so that they remain valid for the new edge set. For the usual combinatorial template cuts, and even for the non-template *local cuts* studied by Applegate et al. (see Chapter 11 of [2]), these needs are met nicely by a *hypergraph* representation of cuts, with a slightly more complex version of this approach employed for domino parity inequalities. While theoretically possible, applying this approach to track Gomory cuts would be a nightmare of computation and bookkeeping. Thus, for most of the TSPLIB instances considered in [6], the authors run their code "on the MIPs obtained after fixing variables to zero by reduced cost." That is, the lower bound initially computed by Concorde allows enough edges to be eliminated from consideration so that a full set of edges may be stored in the core LP, with no need for further pricing or edge generation. They write that

> [in] the cases of the two largest instances, pla33810 and pla85900, however, a large number of variables still remains after reduced-cost fixing; therefore we use an MIP with only a subset of the remaining variables, and hence the final bounds are not necessarily valid for the original TSP.

In either case, Table 1 of [6] shows that the number of variables is on the order of $2n$ to $10n$ for n city TSP instances. The upper range of $10n$ is outside of the ideal target ranges discussed in Chapter 12 of [2], but still of reasonable practical size.

In Camargue I adopt a similar approach: Gomory cuts are only applied to sparse instances, or instances where enough variables have been eliminated to consider the full edge set explicitly.

4.3.2 Computation

We now engage in a limited computational study surrounding the use of Gomory cuts in primal cutting plane computation, aiming only to examine the relative density of primal and standard cuts. Computational results incorporating Gomory cuts into the overall solution process are reported in Section 8.3.

The experiment proceeds as follows. For all TSPLIB [28] instances of size between 2,000 and 15,000 nodes, we do a run of Camargue's pure primal cutting plane loop with a sparse edge set, along with separation routines for subtours, blossoms, block combs, and simple DP inequalities. That is, no Gomory cut separation is applied during the solution process. Once no further cuts are found, we invoke primal Gomory cut separation on the final fractional LP solution found. Thus, the Gomory cut separation algorithm has a relatively large number of constraints at its disposal. We compare average densities in Figure 4.4, plotting with a logarithmic scale. The density of a cut is the number of its nonzero entries relative to the number of columns in the edge set, and the averages are taken over all cuts found. Note that no primal Gomory cuts were found for d2103, pr2392, fl3795, and rl5934; these instances are omitted from the table.

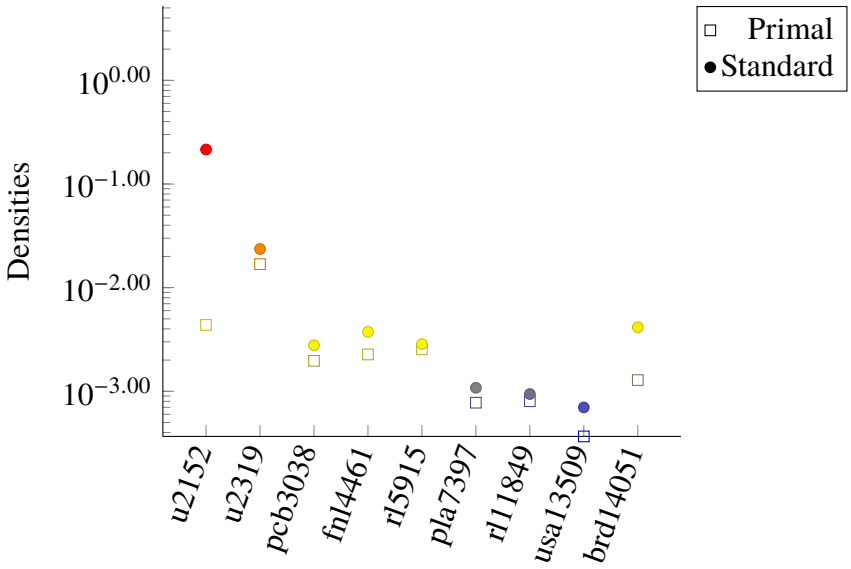


Figure 4.4: Comparing densities of safe Gomory cuts.

Some remarks are in order. While primal cuts are always more sparse on average than standard cuts, it should be noted that almost all of the instances had *extremely* sparse cuts, primal or otherwise. To give an idea, these average densities imply cuts with at most 10-60 nonzero entries. The exception, perhaps counter-intuitively, is the smallest instances tested: with u2152 the average density of cuts found was 0.215320, which is quite dense.

On the whole, it may be tempting to conjecture that restricting our attention to primal Gomory cuts works as a safeguard against adding extremely dense cuts to the LP. The results in Figure 4.4 lend some support to this idea, but I suspect it has more to do with the decision to only aggregate tableau rows for fractional basic structural variables.

4.4 Concluding Remarks and Directions for Further Research

At this point we have completed a review of almost all the separation routines employed in Camargue. The main exception is simple domino-parity inequality separation, which will be discussed in the next chapter. In addition to connected component SECs and odd component blossom heuristics, Camargue offers the option to use several other standard separation algorithms, all as implemented by Applegate et al. in their Concorde TSP solver. These include

- block comb separation (see Section 7.2 of [2]);
- cut metamorphoses: tightening cuts in the LP and deriving new cuts from combs (see Chapter 10 of [2]); and
- the non-template local cuts (see Chapter 12 of [2]).

Each of these routines are highly effective standard separation algorithms, adapted to the primal case simply by checking inequalities found for tightness at the current tour. There is not much interesting to add on this matter. Indeed one could say for any template of cut that it would be interesting to try and adapt standard separation routines to the primal case, although for the most part I do not have any further useful ideas in these areas. Thus, let us now mention two more salient topics which have been omitted from discussion.

4.4.1 Shrinking Rules

One topic which has not been mentioned at all in this thesis is that of *safe shrinking rules*. Shrinking procedures are a key ingredient in cutting plane solution of large-scale TSP instances (see e.g., Section 5.9 of [2]). Given a template $\mathcal{F}$ of cuts, and a support graph G^* in which to search for cuts of type $\mathcal{F}$, the idea is to develop rules by which the node and edge count of G^* can be reduced (usually by contracting nodes and edges), while guaranteeing that no violated cuts will be missed. Such a shrinking rule is called *template-safe* for the template $\mathcal{F}$. Given the importance of such rules to standard separation and dual fractional TSP computation, it seems that they could be equally useful in the primal case.

It is unclear how such rules should be adapted, though, and in what way they should be adjusted to take account of the tour vector $\bar{x}$. To give an idea, consider a path in G^* such that every edge e in the path has x^*_e equal to one. Applegate et al. [2] show that such paths can be safely contracted into a single edge, and this procedure is very useful in their implementation of separation routines which require computing minimum cuts on large support graphs. But of course, the support graph induced by a tour vector is nothing but a path of edges with value one.

When discussing exact primal blossom separation, the only novel approach mentioned for Camargue was a way to parallelize the minimum cut computation. This is a helpful, practical improvement which helps circumvent the expense of computing $|E^*|$ minimum cuts in support graphs for large-scale TSP instances. In my experience with Camargue, this and other safeguards have been essential to ensure that reasonable amounts of time are spent on exact blossom separation. A much more effective approach, though, would be to shrink the support graph on which the minimum cuts are computed.

In the next chapter, we will show how simple domino-parity cut separation on large-scale TSP instances is greatly enhanced by applying a rule which is somewhat in the spirit of support graph shrinking, although its actual workings are quite different.

4.4.2 Chain Constraints

When reviewing the work of Padberg and Hong [25] in Section 2.3.2, we mentioned, in passing, the use of *chain* constraints in their study. Chain constraints appear to be completely absent in TSP computation nowadays; and indeed Padberg and Hong themselves said of such constraints in 1980 that "[they have] yet to be studied in detail from a theoretical point." Chain constraints have a bit of an odd structure, resembling comb inequalities with a nested inner handle and some disconnected teeth. We explain presently.

Consider a family of nonempty, proper subsets $S_0, S_1, \ldots, S_k$ of the vertices of a TSP instance. Moreover, suppose that there are indices p, k with p less than k such that

- $S_i \cap S_0 = \varnothing$ for $i = 1, \ldots, p$; and
- $S_i \cap S_0 \neq \varnothing$ and $S_i \setminus S_0 \neq \varnothing$ for $i = p+1, \ldots, k$; and
- $S_i \cap S_j = \varnothing$ for $i = 1, \ldots, k$.

Moreover, suppose there is a subset R of S_0 such that $|R| = p$ and $R \cap S_i = \varnothing$ for $i = 1, \ldots, k$. And finally, let $\gamma(S)$ denote the set of edges with both ends in S for a vertex subset S, and write $\gamma(S : T)$ for the edges with one end in S and one end in T. Then the chain constraint for the family of subsets $R, S_0, S_1, \ldots, S_k$ reads

$$\sum_{i=0}^{k} x(\gamma(S_i)) + \sum_{i=1}^{p} x(\gamma(R : S_i)) \leqslant |S_0| + |R| + \sum_{i=1}^{k} (|S_i| - 1) - \lceil (k - p + 1)/2 \rceil.$$

This description is a mouthful; we try to convey some intuition with a schematic in Figure 4.5. Here we see the comb-like intersection of the sets $S_{p+1}, \ldots, S_k$ with S_0; the nesting of R in S_0, and edges between $S_1, \ldots, S_p$ and R to indicate the terms $x(\gamma(R : S_i))$ in the summation above.

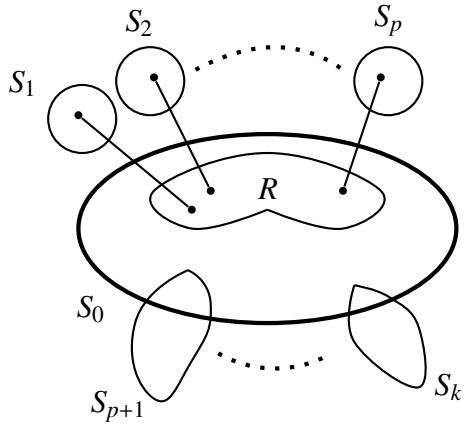


Figure 4.5: A chain constraint schematic.

Padberg and Hong provide a loose description of an algorithm which can be used to separate either comb constraints or chain constraints in an integral subtour vector under the assumption that no primal subtour cuts were found. Indeed it is an interesting bit of TSP polytope trivia to note that subtour constraints are *not* the only constraints which can be violated in an integral solution vector. Of course, this is of little interest in the standard case because it is always easier to just add connected component cuts. In the primal case, though, it provides an appealing fall-back option before allowing non-tight cuts in the relaxation.

Using Padberg and Hong's notation, suppose x^1 is the current tour and x^2 is a non-tour pivot from x^1. They write the following.

> In order to use the above results to cut off fractional vertices x^2 we represent x^2 as a convex combination of two integer vertices (if possible) and attempt to generate constraints ... according to the above proceeding. If the attempt is successful, we check whether or not the fractional vertex x^2 can be cut off this way.

Thus, the authors try to place x^2 on a line segment between x^1 and an integral subtour vector x^3; then they apply their separation routine using x^1 and x^3 as input, adding any cuts found if they do in fact cut off x^2. Padberg and Hong do not say how they try to represent x^2 as a convex combination, although they probably were employing heuristic means of some sort. Indeed, testing whether some vector x^2 is a convex combination of a tour and an integral subtour vector is a restricted form of separating over the fractional 2-matching polytope. An implementation of these separation routines would need to fill out these, and some other more minor details.

The meaning of the "chain" in chain constraint may be a bit unclear. The Padberg-Hong separation algorithm describes constructing the symmetric difference graph $G_S = (V_S, E_S)$ of a

tour vector x^1 and a subtour vector x^2. Here E_S is the set of edges such that $x^1_{ij} \neq x^2_{ij}$, and V_S is vertices incident with edges in E_S. Then,

> [t]he collection of edges in [sic] $E_C = \left\{(i,j)|x^1_j = x^2_j = 1\right\}$ [sic] from "chains" connecting the nodes in V_S.

Figure 4.6 illustrates this idea with the vectors from Figure 4.2, showing edges in E_C in black and edges in E_S highlighted in gray. From this illustration it would appear Padberg and Hong are

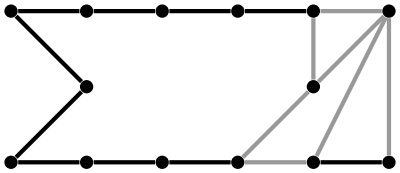


Figure 4.6: The edge sets E_C and E_S for the vectors in Figure 4.2.

using "chain" to mean path or walk, or something to that effect. In Figure 4.6 there are precisely two black-shaded paths, both of which join vertices in V_S.

I mention their notation and turns of phrase only to point out a connection between some more well-known templates of cuts which *are* still used in TSP computation today. In Section 10.3 of [2], Applegate et al. describe the *path* and *star* inequalities, both of which are characterized by nested handle sets. Interestingly, Applegate et al. do not describe these templates as classes of cuts to be separated in their own right. Rather, they may be generated by aggregating and modifying existing inequalities which have already been generated. A path inequality, for example, is created by summing two comb inequalities with nested handles and the same set of teeth. Subject to certain intersection properties on the handles and teeth, it can be possible to modify the coefficients of such an inequality to obtain a stronger cut. Although the intersection properties do not appear to coincide with those for Padberg and Hong's chain constraints, it would be interesting to re-examine chain constraints under this lens. The Padberg-Hong chain separation routine is of complexity comparable to odd-component blossom separation, so it may be useful (in the standard or primal case) to know that such inequalities may be separated in this fashion.

Chapter 5

Simple Domino-Parity Inequalities

This chapter is concerned with primal separation of simple domino-parity (simple DP) inequalities. We will consider an implementation of the exact standard separation algorithm of Fleischer, Letchford, and Lodi [10], with enhancements that Letchford and Lodi [20] describe for the case of primal separation. Of the primal TSP separation algorithms presented by Letchford and Lodi in [20], this algorithm is the only one where there is no asymptotic speedup compared to the standard algorithm; rather the prize is a considerable simplification of an initial subroutine which generates candidate inequalities for the separation algorithm.

In my research I have used Letchford and Lodi's primal simplification to search for candidate inequalities using the exact primal SEC separation routine of Applegate et al. We discussed this algorithm in Section 4.1.1, and it will be reviewed in this context in Section 5.2 . Using this, along with a pre-processing step aided by a simple data structure, asymptotic speedups are obtained in the candidate generation step and in various subroutines related to organizing and paring down the collection of candidates. These enhancements can (and will) be proven to result in the loss of no simple DP inequalities that would otherwise be detected, while also dramatically reducing the memory footprint of the algorithm. Some other heuristic enhancements, particular to the primal approach, are also reported.

The final discovery of my research in this area is a heuristic implementation choice which might charitably be described as a semi-exact algorithm. Say n is the number of nodes in a given TSP instance and m is the number of edges in the support graph for a given solution. The actual separation phase of the Fleischer et al. algorithm involves computing an odd cut in an auxiliary graph with $O(n)$ nodes and $O(m)$ edges. More specifically, the constant factor for the number of nodes is strictly greater than one, and in practice may range around 1.5 to 4. This proves to be a severe bottleneck, with the vast majority of the computation time being spent computing a Gomory-Hu cut tree. I prove that it is possible to obtain the same cuts by considering only

partitions of the auxiliary graph, describing a heuristic method for choosing these partitions.

The rest of this chapter proceeds as follows. We begin by reviewing some of the theory behind simple DP inequalities, looking at the results which underlie the Fleischer et al. algorithm. We then proceed in sections corresponding roughly to subroutines of this algorithm, providing an outline of the Fleischer et al. and Letchford-Lodi approach, together with a more detailed description of enhancements which are regarded as original research in this thesis. It will be clearly indicated whether these enhancements do or do not provide an asymptotic speed improvement, and implementation choices that compromise the theoretical correctness of the exact separation algorithm will be clearly identified as such. Allowing for these theoretical deviations, the work in this chapter may constitute the first computational study of the Fleischer et al. [10] separation algorithm. This claim will be put into context at the end of the next section, with a review of some extant literature concerning domino parity computation.

5.1 Terminology and Literature Review

The exact separation algorithm of Fleischer et al. is a special case of the work of Caprara and Fischetti [5]. Caprara and Fischetti describe a general class of valid integer programming cuts, the $\{0, \frac{1}{2}\}$-*Chvátal-Gomory cuts*, or simply $\{0, \frac{1}{2}\}$-cuts. Given a linear system $Ax \leqslant b$, a $\{0, \frac{1}{2}\}$-cut takes the form $\lfloor \lambda^T A \rfloor x \leqslant \lfloor \lambda^T b \rfloor$ for some λ such that $\lambda^T A$ is integral and each entry of λ is either 0 or $\frac{1}{2}$. An inequality in the system is *used* if the corresponding entry of λ is $1/2$. Caprara and Fischetti show that in a particular case, the $\{0, \frac{1}{2}\}$-cuts can be separated in polynomial time, using an algorithm which is a thoroughly vast generalization of the Padberg and Rao [26] algorithm for exact blossom separation.

Fleischer et al. define a simple DP inequality to be one that can be derived as a $\{0, \frac{1}{2}\}$-cut by aggregating the degree equations and the *simple tooth inequalities*. For a subset S of vertices, let $\gamma(S)$ denote the set of edges with both ends in S. For another subset R of vertices, let $\gamma(R : S)$ denote the set of edges with one end in R and one in S, and relax this notation to $\gamma(i : S)$ in the case that R is a singleton $\{i\}$ contained in V. Let $S \subset V$ such that $1 \leqslant |S| \leqslant |V| - 2$ and let $i \in V \setminus S$. Then the simple tooth inequality with root i and body S is

$$2x(\gamma(S)) + x(\gamma(i : S)) \leqslant 2|S| - 1.$$

This is a convenient time to introduce a pet example which we shall use to develop terms and intuition throughout this chapter. Figure 5.1 shows a subtour polytope solution x^* for a TSP instance on nine cities. (Fleischer et al. [10] use this exact instance as Figure 5 of their paper.)

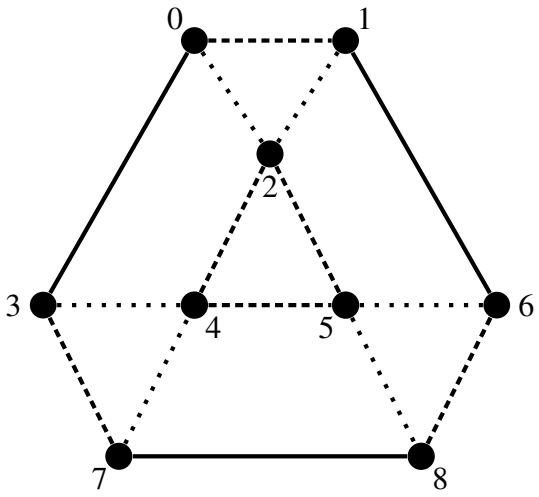


Figure 5.1: Illustration of a vector fractional vector x^* which is in the subtour polytope. Solid lines have weight one, dashed lines have weight $\frac{2}{3}$, dotted lines have weight $\frac{1}{3}$, and all other edges have weight zero.

Example 1. Let us identify some teeth from Figure 5.1. Let $i = 4$ be the root, and let $S = \{2, 5\}$ be the body. Then we can compute the simple tooth inequality with root i and body S:

$$2x(\gamma(S)) + x(\gamma(i : S)) \leqslant 2|S| - 1,$$

which expands to

$$2\left(x^*_{2,5}\right) + x^*_{4,2} + x^*_{4,5} \leqslant 2|S| - 1.$$

This inequality has left-hand side $\frac{8}{3}$ and right-hand side 3, so it is satisfied with slack $\frac{1}{3}$. Teeth may also have singletons as bodies: the tooth with root 7 and body $\{8\}$ gives a simple tooth inequality with slack zero, because $x^*_{7,8} = 1$. Similarly, the tooth with root 3 and body $\{4\}$ gives slack $\frac{2}{3}$. //

As a shorthand, we can denote the tooth with root i and body S by (i, S).

If $R \subset V$, recall we can write the SEC associated to R as $x(\gamma(R)) \leqslant |R| - 1$. In fact, the simple tooth inequality for a tooth with root i and body S is nothing more than the sum of the SECs for S and $\{i\} \cup S$. This shows that simple tooth inequalities are valid for the TSP polytope in general.

The simple DP inequalities are a proper generalization of the *simple comb inequalities*, which are a better known template of TSP cut. Simple comb inequalities are defined by a subset H of V, the handle, and a family of pairwise disjoint subsets $\{T_1, \ldots, T_p\}$ of V with p of odd cardinality at least three, the teeth. We require that both $T_i \cap H$ and $T_i \setminus H$ are nonempty for all i and, moreover, that at least one of $|T_i \cap H|$ and $|T_i \setminus H|$ is one. Then

$$x(\gamma(H)) + \sum_{i=1}^{p} x(\gamma(T_i)) \leqslant |H| + \sum_{j=1}^{p} |T_j| - (3p+1)/2$$

is a valid, facet-defining inequality for the TSP polytope. When we say that the simple DP inequalities are a generalization of the simple comb inequalities, we mean that the simple comb inequalities may be realized as $\{0, \frac{1}{2}\}$-cuts using the degree equations and simple tooth inequalities. Fleischer et al. prove this in [10], and also exhibit a simple DP inequality which is not a simple comb inequality. Moreover, quoting Naddef and Wild [22], the authors observe that simple comb inequalities are the *only* facet-inducing simple DP inequalities, although they induce faces of high dimension in general.

Example 2. Using the vector x^* from Figure 5.1, define the handle set $H = \{3, 4, 7\}$. Consider the following simple teeth and their associated simple tooth inequalities. We have $T_1 = (7, \{8\})$, giving $x_{7,8} \leqslant 1$; and $T_2 = (3, \{0\})$, giving $x_{0,3} \leqslant 1$; and finally $T_3 = (4, \{2, 5\})$, giving

$$2x_{2,5} + x_{2,4} + x_{4,5} \leqslant 3.$$

Summing the degree equations for all vertices of the handle gives

$$2x(\gamma(H)) + x(\delta(H)) \leqslant 2|H|,$$

which expands to

$$2\left(x_{3,4} + x_{3,7} + x_{4,7}\right) + x_{0,3} + x_{2,4} + x_{7,8} \leqslant 6.$$

Take the handle sum and add to it each of the simple tooth inequalities T_1, T_2, T_3 to get

$$2\left(x^*_{0,3} + x^*_{2,4} + x^*_{2,5} + x^*_{3,4} + x^*_{3,7} + x^*_{4,5} + x^*_{4,7} + x^*_{7,8}\right) \leqslant 11.$$

This expands to $2 \cdot \frac{16}{3} \leqslant 11$. And recall that the final step in deriving a $\{0, \frac{1}{2}\}$-cut is to divide the coefficients and right-hand side by two, and then round down. The right-hand side gets rounded down to 5, and the left-hand side is $5 + \frac{1}{3}$, so x^* violates the corresponding simple DP inequality by a third. By checking tooth and handle intersections, it is easy to see that this simple DP inequality is a simple comb inequality. //

Now let us give a brief overview of the Fleischer et al. algorithm, thereby outlining the remainder of this chapter. In the first phase, we search for a collection of candidate simple teeth: teeth which could potentially be used in the derivation of a $\{0, \frac{1}{2}\}$-cut that is a simple DP inequality. These are precisely the simple tooth inequalities with slack less than one. Note that all teeth considered in Example 1 are candidate teeth, but the tooth with root 2 and body $\{8\}$ is not a candidate because the associated inequality has slack one.

There are a great number of candidate tooth inequalities; Fleischer et al. get a polynomial bound on this number by insisting that their separation algorithm only be applied to LP solutions whose support graph is in the subtour polytope. (The algorithm is a non-starter otherwise, and certainly one would rather obtain subtour inequalities for considerably less computational effort if they were available.) Then, some reduction criteria may be applied to eliminate the number of candidate teeth even further.

These candidate teeth are used to generate a weighted auxiliary graph, the *witness* graph. The witness graph is constructed first by defining a rooted tree, and then by joining branches of this tree by certain edges. Odd cuts in this graph of sufficiently low weight correspond to violated simple DP inequalities in the support graph; edges in a cut correspond to inequalities from the original system to which $\{0, \frac{1}{2}\}$ aggregation and rounding shall be applied.

Among candidate simple teeth, Fleischer et al. distinguish between teeth that are *light* and *heavy*. Light teeth are those with slack less than one half, and heavy teeth are teeth which are not light but still have slack less than one. Returning again to Example 1, the tooth with root 3 and body $\{4\}$ is heavy but all others mentioned are light.

Their algorithm first searches for a violated simple DP in which all used teeth are light, and then searches for one in which one of the used teeth is heavy. In my implementation I eschew the heavy tooth phase of the algorithm entirely: it involves modifying the witness graph $O(m)$ times, computing a fresh Gomory-Hu tree and odd cut on each of these graphs. In light of this, and for clarity of exposition, I do not distinguish between light and heavy teeth in the discussion that follows. In the interest of generality I will speak of searching for tooth inequalities with slack less than one, and creating a list of such inequalities sorted by root. But in my actual implementation, I am concerned only with teeth having slack less than one half. To apply the full force of the algorithm we would maintain separate root-organized lists for light and heavy teeth. The enhancements I describe, and the speedups obtained, heuristic or asymptotic, are unchanged.

Previous Studies

At the beginning of this chapter, and in the abstract of this thesis, this work was claimed as the first computational study of the separation routine of Fleischer et al. [10]. Since the Camargue

implementation mixes heuristic and exact approaches, it seems appropriate to review some other literature regarding TSP computation and domino parity inequalities.

Far and away the most successful study is that of Cook, Espinoza, and Goycoolea [7], who implement heuristic (general) domino parity separation in a manner suitable for large-scale TSP instances. We shall have occasion in Section 5.3.2 to review their work in greater detail, using it as motivation for implementation choices in Camargue. So for now, the short story is that their implementation lead to the first reported optimal solution to the TSPLIB [28] instance pla33810. The work of Cook et al. [7] is reviewed by Applegate et al. in Section 9.3 of [2]. In this chapter, Applegate et al. also mention briefly the simple DP separation algorithm of Fleischer et al. which is our current object of interest. Applegate et al. say that

> [t]he Fleischer et al. algorithm is a tour de force, involving a wide range of methods from graph theory, combinatorial optimization, and data structures. To date, no implementation of the method has been studied.

Explicitly, the book [2] was published in 2006. And again, we emphasize that Cook et al. [7] and Applegate et al. [2] were concerned with the separation of general domino parity inequalities, whereas we are concerned presently with the simple domino parity inequalities.

As regards the simple DP inequalities, some interesting results are reported in Section 5.2 of the PhD thesis of Torsten Inkmann [13] (released in 2008), using *tree decompositions* of graphs. Inkmann reports directly on the Fleischer et al. algorithm, echoing Applegate et al. in his remarks about the lack of computational studies, and of the algorithm's conceptual complexity. Inkmann's approach has some common ground with that adopted in Camargue, so a more detailed review is merited.

Essentially, Inkmann presents a heavily modified implementation of the Fleischer et al. algorithm, suitable for use on support graphs with bounded tree width. In such cases, a tree decomposition can be used to speed up the computation of near-minimum cuts for finding candidate teeth, and to facilitate the computation of odd cuts to find simple DP inequalities themselves. In particular, Inkmann observes that, as written, the Fleischer et al. algorithm only returns a *single* violated simple DP inequality by finding one minimum odd cut, as opposed to some or all odd cuts below a certain weight. His tree decomposition approach allows for multiple cuts to be returned in a faster, simpler manner. Inkmann reports favourable computational results on TSPLIB [28] instances of size from 1,000 to 10,000 cities, with a separation routine integrated into the Concorde [2] TSP solver. He also reports, however, that higher tree widths were encountered with some of these instances, causing the implementation to fail due to lack of memory.

5.2 Finding Candidate Teeth

Before anything else, we must find teeth that can be aggregated to generate simple DP inequalities. This is a principle computational bottleneck in the separation algorithm, on par with the computation of odd cuts and Gomory-Hu trees during the separation phase. It is important that we find the teeth quickly, and that we eliminate as many from consideration as possible in a way that will not cause the algorithm to miss a violated simple DP inequality. Moreover, Letchford and Lodi write in [20] that "the bottleneck of this operation is actually storing the sets (which may take $O(n^3|E^*|)$ space to represent)", where E^* denotes the edge set of the support graph. Thus, a further desirable property is to cut down on this storage overhead in a way that allows us to eliminate teeth from consideration before ever having to store them in the first place. This is accomplished by developing a categorization system for simple teeth, using a linear ordering property that is particular to the primal case. We pursue a theoretical development in this direction, at times complementing and extending the theory developed by Fleischer et al. The result is a fast algorithm with light memory overhead that exploits the nature of the primal separation problem to a very high degree.

Let us now commence with a review of the approach of Fleischer et al., and the primal enhancements proposed by Letchford and Lodi. We will borrow terminology from them where appropriate, emphasizing jumping-off points that will be used in the development of faster, more efficient algorithms exposed later in this section.

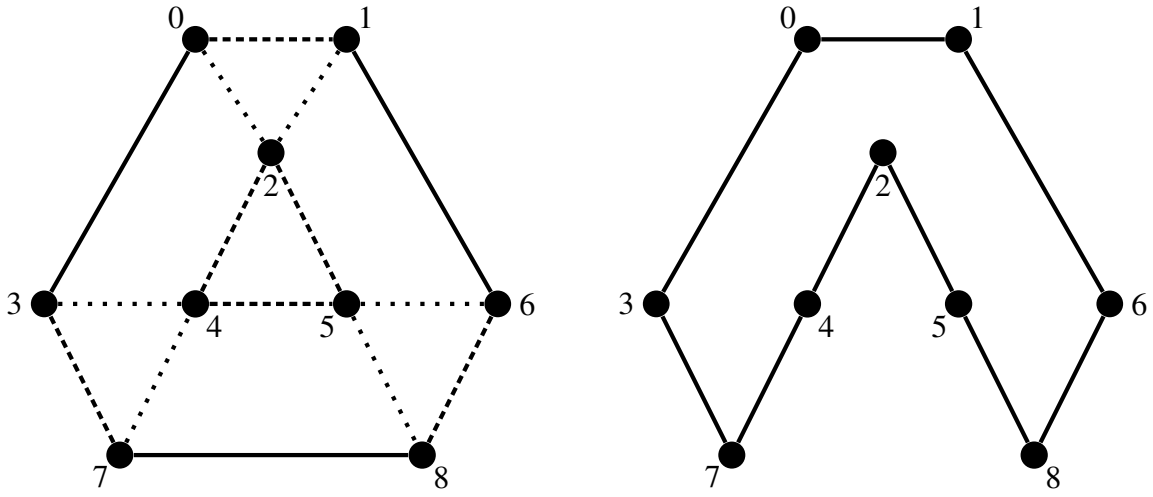
5.2.1 The Standard Approach, and Primal Enhancements

Let G^* denote the support graph of a feasible solution x^* to an LP relaxation of the TSP. Suppose G^* has vertex set V of size n, with edge set E^* of size m. Assuming that G^* is in the subtour polytope, Fleischer et al. show that there are $O(n^3)$ distinct candidate teeth, and these can be found in time $O(nm(m + n \log n))$. The approach is to search for cuts R in G^* such that the slack of the SEC for R is less than one half. If x^* is in the subtour polytope, they show there are $O(n^2)$ such sets; each such R is a candidate to be either S or $\{i\} \cup S$ in a simple tooth inequality with root i and body S, giving $O(n^3)$ inequalities total. The authors propose to find these candidate sets using a near-minimum cut algorithm which runs in time $O(nm(m + n \log n))$.

But each tooth found shall correspond to a vertex in the witness graph: without taking measures to eliminate teeth from consideration, Fleischer et al. describe a run time which "though polynomial, is totally impractical." They propose two criteria which may be applied to eliminate teeth from consideration, each of which runs in time $O(n^3m)$. This dominates the run time of the minimum cut computation, and is regarded as the asymptotic bound for candidate tooth generation.

And indeed, in [20], Letchford and Lodi claim $O(n^3m)$ as the running time of the candidate generation phase. They say that they "have not been able to reduce the asymptotic running time of this algorithm in the primal case, [but] the first step can be simplified considerably." Their insight is that essentially we are just analyzing SECs in the support graph, and, as we saw in Section 4.1.1, these take a convenient, highly restricted form in the primal case. Write (i, S) for the simple tooth with root i and body S, and suppose the vertices of G^* have been permuted so that $1, ..., n$ describes the sequence of the current best tour. Letchford and Lodi use two simple lemmas to show that we need only consider simple teeth of the form (i, S) where S is a nonempty continuous segment of the tour $1, \ldots, n$ and i is drawn from $V \setminus S$. Hence i occurs either directly before or after an endpoint of the segment, or is not adjacent to either endpoint of the segment. With this observation, the candidate teeth can be found "by scanning through all i and all sets S satisfying the conditions ... this can be performed in $O(n^2m)$ time." Here they also mention the storage overhead of $O(n^3m)$ to store the sets.

To develop some intuition for the nature of primal candidate teeth, we pair the solution x^* from Figure 5.1 with a tour $\bar{x}$ associated to the primal separation problem. In Figure 5.2a we reproduce Figure 5.1, and in Figure 5.2b we depict a tour vector $\bar{x}$.



(a) The vector x^*.

(b) The tour vector $\bar{x}$.

Figure 5.2: A fractional subtour polytope vector x^* and a tour $\bar{x}$.

Example 3. Having identified candidate simple teeth in Example 1, let us identify some *primal* candidate simple teeth. The vector x^* from Figure 5.2a will be the support graph solution vector, with the tour $\bar{x}$ from Figure 5.2b as the tour vector. The tooth with root 4 and body $\{2, 5\}$ is a

primal candidate tooth because $2, 5$ is a subsequence of the tour nodes, and 4 appears directly before 2 in the tour. Any tooth with a singleton body is a primal candidate also: we require that i appears directly before or after an endpoint of S, or that i is adjacent to no endpoint of S. This is always vacuously true if $S = \{v\}$ for some vertex v.

In Example 2 we computed a simple comb inequality on Figure 5.2a. In fact, this simple comb inequality is a primal simple comb inequality for the tour $\bar{x}$ from Figure 5.2b. To see this, note that the comb inequality was derived with the teeth $(3, \{0\})$, $(7, \{8\})$, and $(4, \{2, 5\})$, all of which are primal candidate simple teeth. A bit of computation, along the lines of that performed in Example 2, will show that the comb inequality is in fact tight at the vector $\bar{x}$.

For a simple negative example, note that the tooth with root 3 and body $\{4, 5\}$ is a candidate simple tooth which is *not* a primal candidate simple tooth. The associated simple tooth inequality is

$$2x^*_{4,5} + x^*_{3,4} \leqslant 2(2) - 1.$$

It has left-hand side $\frac{5}{3}$ and right-hand side 3, so it is satisfied with slack $\frac{1}{3}$. But $\{4, 5\}$ is not a segment of the tour in Figure 5.2b, so it does not satisfy the primal candidate property. //

A minor point must be addressed: all discussions remaining in this chapter shall assume that the support graph G^* lies in the subtour polytope. In the standard case, this information would likely be obtained for free as a matter of course in normal cutting plane generation. In TSP computation, a reasonable approach is to have "[separation] routines ordered according to rough estimates of their computational requirements" (Applegate et al., Chapter 16 of [2]). In the standard case one would generally search for SECs early in the cut generation process, beginning with connectivity heuristics and then searching exactly for a minimum cut in the graph. Thus, failure to find standard SECs indicates that the minimum cut in the support graph has weight two, which characterizes membership in the subtour polytope. As we saw in Section 4.1.2, however, failure to generate primal SECs is necessary but *not* sufficient for membership in the subtour polytope. Thus if SECs are generated we know that simple DP separation cannot proceed, but if not then it still remains to compute a minimum cut in G^*. Following the discussion in Chapter 6 of Applegate et al. [2], the approach in Camargue is to compute the value of a global minimum cut in G^*. Since we are only interested in a numerical value rather than returning minimum cut sets, we apply aggressive shrinking rules to reduce the size of the input graph, and then use a push-relabel minimum cut algorithm. In theory and practice, this step is decidedly not a computational bottleneck, but it should be accounted for as an expense of this separation routine all the same.

5.2.2 An Elimination Criterion

Fleischer et al. provide two elimination criteria for reducing the number of candidate teeth. The first takes time $O(n^3m)$, and leaves $O(m)$ light teeth: the bound is $4m - 2n$ to be exact. The second criterion takes $O(n^3m)$ time as well for an $O(n)$ bound on the number of light teeth, with some more involved uncrossing arguments. The second criterion is not implemented in Camargue, given that the support graphs arising in TSP computation tend to be so sparse that $O(n)$ is not much lower of a target to set than $4m - 2n$. Thus, we will focus only on the first criterion, paraphrased below. Using E^* for the edge set of the support graph G^*, and with x^* as the LP solution used to define G^*:

Lemma 5.2.1 (Lemma 5.5, [10])**.** *Suppose a violated $\{0, \frac{1}{2}\}$-cut can be derived using the tooth inequality with root i and body S. If there exists a set $S' \subset V \setminus \{i\}$ such that*

1. $\gamma(i : S) \cap E^* = \gamma(i : S') \cap E^*$,
2. $|S'| - 2x^*(\gamma(S')) - x^*(\gamma(i : S')) \leqslant |S| - 2x^*(\gamma(S)) - x^*(\gamma(i : S))$,

then we can obtain a $\{0, \frac{1}{2}\}$-cut violated by at least as much by aggregating the tooth (i, S') in place of (i, S), and rounding accordingly.

Lemma 5.2.1 provides a notion by which two bodies S, S' may be considered *equivalent* with respect to a given root. Explicitly, for each i we may define an equivalence relation on subsets of V, declaring S equivalent to S' with respect to i precisely when the first condition holds in the lemma above, i.e., when

$$\gamma(i : S) \cap E^* = \gamma(i : S') \cap E^*.$$

We will then say that S and S' are *root* equivalent when there is some i for which the first condition of Lemma 5.2.1 holds; we may call them *i-equivalent* to emphasize the root i, writing $S \sim_i S'$. Let $\mathcal{S}$ denote the set of all nonempty subsets of V of size at most $|V| - 2$, i.e., all subsets which can be the body of a simple tooth inequality. Then write $\mathcal{S}/\sim_i$ for the set of all equivalence classes on $\mathcal{S}$ with respect to $\sim_i$.

Fleischer et al. point out that

> [the] second condition in the lemma simply says that the slack of the tooth inequality with root i and body S' is not greater than the slack of the tooth inequality with root i and body S.

It is easy to check that the second condition defines a weak ordering on $\mathcal{S}/\sim_i$. For two i-equivalent subsets S and S', write $(i, S') \preceq (i, S)$ to indicate that the tooth inequality for (i, S') has slack less

than or equal to the tooth inequality for (i, S). In other words, $(i, S') \preceq (i, S)$ when the second condition holds in the lemma above.

Thus, Lemma 5.2.1 says that for each equivalence class in $\mathcal{S}/\sim_i$, we keep at most one representative which is minimal with respect to the ordering $\preceq$. This suggests defining the following function prototype.

$$\textsc{Equivalent}(i, S, S') \qquad \text{Returns true if and only if } S \sim_i S'.$$

A straightforward implementation of Equivalent, suitable for the primal or standard case, runs in time $O(n + |\delta(i)|)$: we traverse the elements of S and S', marking elements of an n-length integer array to indicate membership in one, both, or neither of the sets. Then we check the neighbour set $\delta(i)$ to see if there is any vertex which appears in only one of S or S'.

In the next section, we describe a simple pre-processing step which, in the primal case, allows Equivalent to be computed in time $O(|\delta(i)|)$ for a given root i. Then, we describe another very simple data structure which capitalizes on our primal implementation of Equivalent to allow teeth to be eliminated as a constant time subroutine of the candidate generation process. This second step takes $O(m^2/n)$ time and space.

5.2.3 A Pre-processing Step

The discussion of the previous section made no particular note of differences between the primal and standard case. The ideas underlying the algorithms of this section exploit the primal case directly: we use the cyclic ordering of vertices given by the current tour to determine at a glance whether two bodies are equivalent. To explain, we borrow again the notation used by Applegate et al. (see Section 6.3 of [2]) to describe tour intervals: if $1, \ldots, n$ is the tour, let I_{st} denote the vertex subset $\{s, s+1, \ldots, t\}$. Now suppose that we are considering a particular root i with neighbors

$$j_1, \ldots, j_d, \quad \text{where} \quad j_1 < j_2 < \cdots < j_d.$$

Explicitly, this means that $j_1, \ldots, j_d$ are sorted with respect to the ordering of tour vertices.

In computing $\gamma(i : I_{st})$, note that a given neighbor j_k of i will contribute to $\gamma(i : I_{st})$ if and only if $s < j_k < t$: if the indices are arranged thusly then $j_k \in I_{st}$, hence $\gamma(i : I_{st})$ contains the edge (i, j_k); and the converse is clear. Thus, the neighbors $j_1, \ldots, j_d$ provide a partition of the

vertex set into *adjacency zones*. Define

$$\begin{aligned} Z_i(0) &= \{1, 2, \ldots, j_1 - 1\}, \\ Z_i(1) &= \{j_1, j_1 + 1, \ldots, j_2 - 1\}, \\ &\vdots \\ Z_i(k) &= \{j_k, j_k + 1, \ldots, j_{k+1} - 1\}, \\ &\vdots \\ Z_i(d) &= \{j_d, j_{d+1}, \ldots, n\}. \end{aligned}$$

For each i, we want to define a function f_i whose domain is the the vertices $V \setminus \{i\}$, and whose range is adjacency zones of i and actual neighbors of i. That is,

$$f_i(j) = \begin{cases} j_k & \text{if } j = j_k \text{ for some } j_k \text{ in } \{j_1, \ldots, j_d\} \\ Z_i(k) & \text{if } j \in Z_i(k) \text{ for some } Z_i(k) \text{ in } \{Z_i(0), \ldots, Z_i(d)\} \text{ and } j \notin \{j_1, \ldots, j_d\}. \end{cases}$$

We claim that access to the family of functions $f_1, \ldots, f_n$, gives an implementation of Equivalent. To explain, we recall and refine some of the notation for root equivalence classes from the previous section.

Fixing a given tour, write $\mathcal{I}$ for the set of all nonempty tour segments of length at most $|V| - 2$. Thus, $\mathcal{I}$ is a proper subset of the set $\mathcal{S}$ described in the previous section, and it is particular to a tour. Just as we considered the set of equivalence classes $\mathcal{S}/\sim_i$ of vertex subsets with respect to i-equivalence for a root i, we may consider the restricted set of equivalence classes $\mathcal{I}/\sim_i$. That is, $\mathcal{I}/\sim_i$ is equivalence classes of *tour intervals* with respect to i-equivalence.

For each root i with degree d, the elements of $\mathcal{I}/\sim_i$ may be represented unambiguously by an ordered pair of integers (k, ℓ), where

$$1 \leqslant k \leqslant \ell \leqslant d.$$

We say that S belongs to the equivalence class for (k, ℓ) if

$$\gamma(i : S) \cap E^* = \{(i, j_k), (i, j_{k+1}), \ldots, (i, j_\ell)\}.$$

The concept of root equivalent sets is illustrated in Figure 5.3, providing a simple schematic diagram of two sets S and S', nested so that $S \subsetneq S'$. The node at the bottom is some root vertex i, with dashed lines indicating neighbour vertices which lie properly in the set S, and which are present in the support graph edges E^*. Although S and S' are distinct, they are i-equivalent because S' contains no neighbour of i that is not in S.

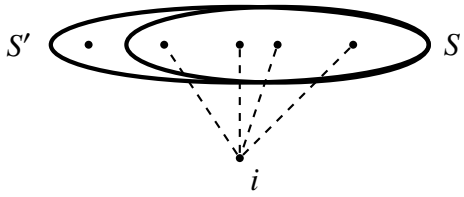


Figure 5.3: Two nested sets S and S' which are i-equivalent.

Thus, k and ℓ represent the start and end indices of neighbours of i contributing to $\gamma(i : S) \cap E^*$. (Note that $k \leqslant \ell$ always because we ignore tour intervals that "wrap around" from n back to 1, but these could just as well be included.) We get the pair (k, ℓ) for an interval I_{st} using a function with prototype

GET-REP(i, I_{st}) Get the ordered pair representing the equivalence class in $\mathcal{I} / \sim_i$ containing I_{st}.

And the result of EQUIVALENT(i, I_{st}, I_{uv}) is true if and only if the ordered pairs

$$\text{GET-REP}(i, I_{st}) \quad \text{and} \quad \text{GET-REP}(i, I_{uv})$$

are equal. Thus, an implementation of GET-REP, and hence of EQUIVALENT, is provided by the functions f_i. The inner workings are just a mild bit of case analysis; we consider some illustrative cases in Example 4.

Example 4. We consider some possible outputs of the value $(f_i(s), f_i(t))$ when considering an interval I_{st} with respect to a root i.

- If $(f_i(s), f_i(t)) = (j_k, j_k)$, then $s = t = j_k$, so the pair is (k, k).
- If $(f_i(s), f_i(t)) = (j_k, Z_i(k))$ then $s = j_k$ and $t \in Z_i(k)$, hence the pair is again (k, k) because no additional neighbours of i lie between s and t.
- If $(f_i(s), f_i(t)) = (Z_i(k-1), Z_i(\ell+1))$, then $s < j_k$ and $j_\ell < t$, hence the pair is (k, ℓ) because k and ℓ are the lowest and highest indices of neighbours of i lying in I_{st}.

Now that we have an idea of how equivalent teeth may be identified, we step back for a slightly more big-picture point of view to see how we will be able to identify and eliminate teeth in place as they are found.

For each root i, we grow and maintain a list $\mathcal{L}_i$ of simple teeth with root i. A reasonable approach to storing members (i, I_{st}) of a list $\mathcal{L}_i$ is suggested by the very notation we use: we

keep the integers i, s, t for the root and body of the tooth, and we keep a floating point number for the slack of the associated tooth inequality. When a new tooth (i, I_{st}) is found, the value Get-Rep(i, I_{st}) allows us to recognize the tooth as a representative of an equivalence class in $\mathcal{I} / \sim_i$. Thus we just need access to some sort of lookup table $\mathcal{M}_i$ which can report the position in $\mathcal{L}_i$ of an existing tooth from the same equivalence class. If $\mathcal{M}_i$ reports that no representative of this class has been found yet, we add (i, I_{st}) to the back of $\mathcal{L}_i$. If a representative exists already, we add (i, I_{st}) to $\mathcal{L}_i$ if and only if it is a strict improvement on the slack of the previous tooth, in which case the old tooth is deleted and (i, I_{st}) takes its place in the list.

Now we just have to get the f_i's, and the lookup tables $\mathcal{M}_i$. The approach is quite straightforward, and we use exceedingly simple data structures for both.

The family of functions f_i are implemented using an adjacency list representation of the support graph G^*. For each node i in G^*, if the adjacency list for i is sorted to conform with the cyclic ordering of the tour, then each f_i can return its result in time $O(|\delta(i)|)$. The implementation is spelled out by the very notation used to define the adjacency zones at the beginning of this section. Suppose that $j_1, \ldots, j_d$ are the neighbours of i, and that $j_1 < \cdots < j_d$. This sorting allows us to define the adjacency zones almost instantaneously, and we can easily enumerate the cases to be considered:

- if $1 \leqslant s < j_1$, then $i \in Z_i(0)$ so $f_i(s) = Z_i(0)$.
- For k from $1, \ldots, d-1$, if $j_k \leqslant s < j_{k+1} - 1$, then $s \in Z_i(k)$, so $f_i(s) = j_k$ if $s = j_k$, or $Z_i(k)$ otherwise.
- If $j_d \leqslant s \leqslant n$, then $s \in Z_i(d)$, so $f_i(s) = j_d$ if $s = j_d$, or $Z_i(d)$ otherwise.

The implementation of the functions f_i in Camargue returns $-k$ for some positive integer k to report that $f_i(s) = j_k$, or some nonnegative integer ℓ to report that $f_i(s) = Z_i(\ell)$. It is clear that this procedure runs in time $O(|\delta(i)|)$. The function Get-Rep is responsible for translating these sentinel values into valid ranges, as sketched in Example 4.

Taking advantage of the fact that $k \leqslant \ell$ for all ordered pairs (k, ℓ) returned by Get-Rep, we store each $\mathcal{M}_i$ as a $d \times d$ upper triangular matrix of pointers to list entries. The coordinate $\mathcal{M}_i(k, \ell)$ will point to the position in list $\mathcal{L}_i$ containing a representative of the equivalence class (k, ℓ); it will be null if none has yet been found. Dereferencing the value of $\mathcal{M}_i(k, \ell)$ allows us to check if a new tooth is a strict improvement on the slack of an old one. Algorithm 5.2.1 describes the particular sorting and allocation steps performed to enable this cooperation.

For each i from the n nodes of G^*, the first step in the **for** loop of Algorithm 5.2.1 allocates

Algorithm 5.2.1: Sorting the support graph and allocating the lookup tables.

Data: An adjacency list representation of a support graph G^* on n nodes and m edges, and the cyclic ordering associated to a tour $\bar{x}$.

Result: A sorted version of G^*, and the lookup matrices $\mathcal{M}_i$ for use representing teeth found.

for *each node* i *in* G^* **do**

Allocate the $|\delta(i)|$-length matrix $\mathcal{M}_i$, with all values initially null

Sort the neighbour list $\delta(i)$ according to the ordering of $\bar{x}$

a block of memory of size $O(|\delta(i)|^2)$. [1] Thus, the amount of space required, and the number of element accesses, is bounded above by the sum of squares of degrees in the graph. In [9], de Caen shows that

$$\sum_{i=1}^{n} |\delta(i)|^2 \leqslant m\left(\frac{2m}{n-1} + n - 2\right),$$

giving a time and space upper bound of $O(m^2/n)$ for the lookup tables.

The other step in the **for** loop is to sort for each i a list of length $|\delta(i)|$. Abbreviating $|\delta(i)|$ as d_i, this can be done in time $O(d_i \log d_i)$ for each i, so the total time required is bounded above by the sum of $d_i \log d_i$ over all nodes i in the graph. A tight bound on this sum does not seem obvious, but it is certainly true that $O(d_i \log d_i)$ is $O(d_i^2)$, so the $O(m^2/n)$ time for the lookup tables is also the asymptotic time bound for Algorithm 5.2.1 as a whole. Moreover, this initialization and sorting can easily be done in parallel.

5.2.4 Finding and Eliminating in Place

Let us now put the adjacency zones and lookup tables to work, using them to help find and pare down a collection of candidate simple teeth. Our goals are threefold:

- Find candidate teeth efficiently;
- keep the collection of candidates small *as they are found*, rather than eliminating from an overgrown collection; and
- reduce as much as possible the effort required to sort the candidates by body size.

[1] An efficient representation of an upper triangular matrix as a ragged array can reduce this to precisely $|\delta(i)| + \binom{|\delta(i)|}{2}$.

The approach we take is a cooperation between the machinery of the previous section, and an algorithm used to find all segment cuts below a fixed capacity. As mentioned earlier, in a straightforward implementation, one keeps for each node i a list $\mathcal{L}_i$ of simple teeth with root i, arriving at a collection of lists $\mathcal{L}_1, \ldots, \mathcal{L}_n$ which are then each eliminated and sorted. The final outcome of our algorithm will be the same collection of lists, but instead we maintain *during* the candidate generation process that no two teeth in $\mathcal{L}_i$ are i-equivalent.

Letchford and Lodi [17] write that eliminating by equivalence leaves a collection of $O(m)$ light teeth. And Fleischer et al. [10] write that the same criterion reduces the number of heavy teeth to $O(nm)$. Recall that a simple tooth is stored with just three integers and a floating point value, so either way we obtain a considerable improvement on the $O(n^3m)$ space requirement claimed by Letchford and Lodi [20].

The main engine of the candidate generation process is an algorithm of Applegate et al., Algorithm 6.9 in [2]. We shall refer to the algorithm as

ALL-SEGMENTS(α) Find all segment cuts with slack less than α.

In Section 6.3 of [2], Applegate et al. describe an implementation of ALL-SEGMENTS, explaining that the algorithm was developed to generate a repository of sets for use in a comb separation heuristic; so our use case is very much in the same spirit. If C is the number of cuts in the graph with slack less than α, Applegate et al. show that their implementation of ALL-SEGMENTS runs in time $O((m+C)\log n)$. In our case $\alpha = 1/2$, and Fleischer et al. [10] show that C is $O(n^2)$, so the base running time is $O((m+n^2)\log n)$. The approach we describe is implemented as a callback to ALL-SEGMENTS, invoked each time a new segment cut is discovered. We shall describe a callback running in time $O(m)$, giving an asymptotic running time $O((m+n^2m)\log n)$.

To explain how the callback works, we give a bit more detail on the workings of ALL-SEGMENTS. It is an extension of the algorithm described in Section 4.1.1 for finding SECs arising from tour segments. To rephrase slightly the discussion of that section, a search for violated primal SECs is really just a search for segment cuts with slack less than zero, setting $\alpha = 0$ above. Scanning through each left endpoint i from $n, n-1, \ldots, 1$, the algorithm described in Section 4.1.1 would return a single right endpoint t and a cut value β such that the SEC for I_{it} is violated by β, and β is maximal over all t from i to n. The difference in ALL-SEGMENTS is therefore suggested by its name: we scan through left endpoints and return *all* right endpoints (and their associated cut values) giving slack less than α.

The callback works in two modes. We keep a record of the last segment returned by ALL-SEGMENTS for comparison with the newest segment found. If their left endpoints disagree, we enter the initialization step of the callback, cleaning and resetting the auxiliary data structures. Otherwise, we enter a general phase, accumulating more data associated with the traversal of the

current left endpoint. At the end of either phase, we check for candidate simple teeth that may be added to the current lists.

The data structures maintained by the callback are extremely simple. As mentioned, we keep a copy of the last segment found, so that the next segment may have its left endpoint compared against the previous one. (Before calling ALL-SEGMENTS this is initialized to some dummy value.) Additionally, we keep two n-length arrays. The first is `marks`, an array of boolean values used to record set membership: when considering a segment S as the body of a tooth (i, S), valid choices for i are evidently vertices in $V \setminus S$. Thus, `marks[j]` is set to one for all j in S, zero otherwise. The second is `gamma`, an array used to represent values of $x^*(\gamma(i : S))$ for potential teeth (i, S). We pause for a moment to explain the meaning of the array `gamma`.

Supposing x^* is in the subtour polytope and (i, S) is a candidate simple tooth, the simple tooth inequality for (i, S) evaluated at x^* is

$$2x^*(\gamma(S)) + x^*(\gamma(i : S)) \leqslant 2|S| - 1.$$

We claim that if x^* is indeed a candidate simple tooth, then $x^*(\gamma(i : S)) > 0$. To see this, note that slack of the inequality above is

$$2|S| - 1 - 2x^*(\gamma(S)) - x^*(\gamma(i : S)),$$

and if (i, S) is a candidate tooth then its slack is less than one. Thus

$$2|S| - 1 - 2x^*(\gamma(S)) - x^*(\gamma(i : S)) < 1,$$

which may be rearranged to give

$$2|S| - 2x^*(\gamma(S)) - 2 < x^*(\gamma(i : S)).$$

But x^* satisfies the degree equations, so

$$2|S| - 2x^*(\gamma(S)) = x^*(\delta(S)),$$

hence the preceding inequality says

$$x^*(\delta(S)) - 2 < x^*(\gamma(i : S)).$$

Finally, since x^* is in the subtour polytope, this means $x^*(\delta(S)) \geqslant 2$, so $x^*(\gamma(i : S)) > 0$, as claimed. And in fact, we have shown that $x^*(\gamma(i : S))$ must be bigger than $x^*(\delta(S)) - 2$ for (i, S) to be a candidate simple tooth.

Since ALL-SEGMENTS returns intervals I_{st} together with their SEC slack, we have ready access to the value $x^*(\delta(I_{st})) - 2$. The idea is to update `marks` and `gamma` in an incremental fashion,

resetting them to zero when a new left endpoint is encountered. But if we instead go from some segment I_{st} to $I_{st'}$, where $t' > t$, we scan through the neighbours of each vertex j from $t + 1$ up to t'. For each i that is adjacent to such a vertex j, we add x^*_{ij} to `gamma[i]`. At the same time, we use `marks` to keep track of vertices that are now in $I_{st'}$, as their `gamma` values are no longer of interest. To add candidate teeth, we examine the nonzero entries of `gamma`: if `gamma[i]` is bigger than $x^*(\delta(I_{st'})) - 2$, then $(i, I_{st'})$ is a candidate simple tooth. The explicit description of

these procedures is summarized in Algorithm 5.2.2.

Algorithm 5.2.2: Subroutines of the callback to ALL-SEGMENTS.

Data: The support graph G^* of a subtour polytope solution x^*, and its adjacency zone arrays and lookup tables

Function `Initialize`(I_{ss})
- **Data:** A single-element segment I_{ss} with as-yet-unseen left endpoint s
- **Result:** The arrays `marks` and `gamma` initialized for handling further segments with same left endpoint s
- Set all entries of `marks` and `gamma` to zero
- Set `marks[s]` $\leftarrow 1$
- **for** *all neighbours i of s in G^** **do**
  - Set `gamma[i]` $\leftarrow x^*_{is}$

Function `Update`(I_{st}, I_{st_0})
- **Data:** A segment I_{st} with previously seen segment I_{st_0}, and arrays `marks` and `gamma` containing data for I_{st_0}
- **Result:** The arrays `marks` and `gamma` updated for I_{st}.
- **for** *i from $\{t_0 + 1, t_0 + 2, \ldots, t\}$* **do**
  - Set `marks[i]` $\leftarrow 0$
  - Set `gamma[i]` $\leftarrow 0$
- **for** *i from $t_0 + 1, t_0 + 2, \ldots, t$* **do**
  - **for** *all neighbours u of i in G^** **do**
    - **if** *`marks[u]` is zero* **then**
      - Increment `gamma[u]` by x^*_{iu}

Function `Add-Tooth`$((i, I_{st}))$
- **Data:** The list $\mathcal{L}_i$ of teeth with root i, and the lookup table $\mathcal{M}_i$ for root i
- **Result:** The tooth (i, I_{st}) is added to $\mathcal{L}_i$ if appropriate
- Let (k, ℓ) be GET-REP(I_{st})
- **if** *$\mathcal{M}_i(k, \ell)$ is null* **then**
  - Add (i, I_{st}) to the back of $\mathcal{L}_i$, making $\mathcal{M}_i(k, \ell)$ point to the back of $\mathcal{L}_i$
- **else**
  - Let (i, I_{uv}) be the tooth pointed to by $\mathcal{M}_i(k, \ell)$
  - **if** *(i, I_{st}) has strictly lower slack than (i, I_{uv})* **then**
    - Replace (i, I_{uv}) with (i, I_{st}) in $\mathcal{L}_i$.

Note that in the function `Initialize` we are guaranteed that the first interval encountered does indeed have the form I_{ss} because our LP solutions satisfy the degree equations, hence each singleton gives a cut with slack zero.

Bounds on running times of routines in 5.2.2 are straightforward. In `Initialize` we clear the n-length arrays and write an entry of `gamma` for each neighbour of a single vertex; the $O(n)$ to clear the arrays is evidently the main expense. In `Update` we have two segments I_{st} and I_{st_0}, and the difference between t_0 and t is certainly $O(n)$. Thus we zero out $O(n)$ array entries, and examine the neighbour sets of $O(n)$ vertices to increment `gamma`, which again is $O(m)$ by the handshaking lemma. Finally there is `Add-Tooth`, where the pre-processing steps from earlier give us a running time $O(|\delta(i)|)$ for a given root i.

Algorithm 5.2.3: The callback to ALL-SEGMENTS.

Data: A segment I_{st} returned by ALL-SEGMENTS(1/2), its SEC slack α, a previously seen segment $I_{s't'}$, and the array `gamma` of root sums

Result: If any candidate teeth with body I_{st} are found, they are added to corresponding lists if appropriate

Compute the lower bound $\beta = x^*(\delta(I_{st})) - 2$
if *$s \neq s'$ and I_{st} is a singleton* **then**
    Call `Initialize`(I_{st})
else
    Call `Update`$(I_{st}, I_{s't'})$

for *each i such that* `gamma`$[i] > \beta$ **do**
    Call `Add-Tooth`$((i, I_{st}))$

The ingredients of Algorithm 5.2.2 are put together in the routine described in Algorithm 5.2.3. At each invocation, the callback performs one of the subroutines from Algorithm 5.2.2, which is $O(m)$ in the worst case for `Update`. Then, we scan through nonzero entries of the n-length array `gamma`, making $O(n)$ calls to `Add-Tooth`; the handshaking lemma again gives an $O(m)$ running time bound.

Thus the bottleneck of Algorithm 5.2.3 is the calls to `Update`, and the iterated calls to `Add-Tooth`. With ALL-SEGMENTS running in time $O((m + n^2) \log n)$, and calling Algorithm 5.2.3 for each of those n^2 cuts found, we arrive at the promised $O((m + n^2 m) \log n)$ running time, which is $O(n^2 m \log n)$. This is the final bound for the step of candidate tooth generation, as we have shown in the course of our discussion that the elimination criterion of Lemma 5.2.1 has been applied exhaustively in the course of Algorithm 5.2.3. The figure of $O(n^2 m \log n)$ also

asymptotically dominates the $O(m^2/n)$ required in the pre-processing steps.

As a slight digression, recall that in Section 4.1.1 we described how Padberg and Hong [25] wrote a $O(mn)$ primal SEC separation routine which was improved to $O(m \log n)$ by Applegate et al. [2]. This invites a nice parallel to the results in this chapter: recall that Letchford and Lodi [20] claimed a running time of $O(n^3 m)$ for finding and eliminating candidate teeth, which we have improved to $O(n^2 m \log n)$, hence trading an n for a $\log n$ just as in the case of primal SEC separation.

5.2.5 Sorting Teeth

The final step in the candidate tooth generation process is to sort each list of teeth in order of increasing body size. This is done in preparation for building the witness graph, which is the next phase of the simple DP separation algorithm. Thus we are at the end of the candidate phase of the algorithm. We will close this section with a brief description of a simple heuristic speedup which occurs for free as a consequence of using the routine All-Segments to find candidate teeth.

In Section 6.3 of [2], Applegate et al. explain that their implementation of All-Segments uses a heap structure which stores leaves "in the left-to-right order". More explicitly, they explain that for each possible left endpoint s of a tour interval, right endpoints are examined in increasing order from $\{s+1, s+2, \ldots, n\}$. Thus, the segments found by All-Segments are examined and returned as a list of sorted subsequences. More explicitly, for each left endpoint s, if All-Segments returns the segments

$$I_{st_1}, I_{st_2}, \ldots, I_{st_k},$$

then

$$s = t_1 < t_2 < \cdots < t_k \leqslant n.$$

So for each s, the segments with left endpoint s are sorted in order of increasing body size. And All-Segments scans the vertex set from n down to 1, giving the list of sorted subsequences. Since intervals are found in this order, Algorithm 5.2.3 will add (or not add) the segments to appropriate lists in this order, too. From a practical implementation point of view, an adaptive sorting algorithm is likely to perform better at this step. But to be clear, the sorting step accounts for an extremely small fraction of the effort required in generating candidate teeth, or in the algorithm as a whole.

5.3 Building the Witness Graph

With a collection of candidate teeth in hand, the next step is to try and use them to find a violated simple DP inequality. This is accomplished by building the so-called *witness* graph. The insight of the Fleischer et al. separation procedure is to show that odd cuts in the witness graph of weight less than one correspond to violated simple DP inequalities.

In this section we proceed as follows. First, we will provide a fairly careful summary of the procedure for constructing the witness graph. This is for the sake of clarity, and to motivate the following sections, in which we identify a heuristic shrinking-style operation which proves vital for efficiently running the separation algorithm on larger instances.

5.3.1 The Algorithm

Recall that a simple DP inequality is one which is obtained by applying $\{0, \frac{1}{2}\}$-rounding to the degree equations, candidate simple tooth inequalities, and nonnegativity inequalities on edges. Thus, the witness graph is responsible for encoding the interplay of all three of these in such a way that odd cuts correspond to simple DP inequalities. We will now describe the algorithm for constructing a witness from a collection of candidate teeth. Our development will proceed much along the same lines as Fleischer et al. [10], who describe the witness graph as a particular case of the graphs described in Caprara and Fischetti's paper [5] on $\{0, \frac{1}{2}\}$-cuts.

The witness graph is built in two phases. In the first, we join a collection of subtrees to build a tree. This tree does the work of encoding the degree equations and simple tooth inequalities, essentially using edges to represent a containment relationship between simple teeth. Next, we join subtrees by edges which represent the nonnegativity inequalities. We describe the tree procedure presently. Then, we will describe the added nonnegativity edges, with a bit more motivation on the significance of the nonnegativity inequalities.

The tree is built according to Figure 8, Lemma 4.1, and Theorem 4.1 of Fleischer et al. [10]. We paraphrase their discussion here, borrowing some notation.

Consider one of the lists $\mathcal{L}_i$ from the previous section. That is, $\mathcal{L}_i$ is a collection of candidate simple teeth with root i. Define a central root vertex v^* which is joined to a vertex v_0 by an edge of weight zero. This edge represents the degree equation for node i. Now, let (i, S_1) be a tooth from $\mathcal{L}_i$. If there is no other tooth (i, S_2) in $\mathcal{L}_i$ such that $S_1 \subset S_2$, then we join v_0 to a vertex v_1 by an edge with weight equal to the slack of the simple tooth inequality for (i, S_1); the edge represents that inequality. Thus, all teeth whose bodies are inclusion-wise maximal from $\mathcal{L}_i$ are joined by an edge to v_0. Next, suppose that $(i, S_2) \in \mathcal{L}_i$ and $S_2 \subset S_1$. Suppose moreover that there is no other tooth body S_3 from $\mathcal{L}_i$ such that $S_2 \subset S_3$ but $S_3 \subset S_1$. In that case, we join a vertex v_2 to v_1 by an edge with weight equal to the slack of the simple tooth inequality for (i, S_2); this edge represents

the simple tooth inequality for (i, S_2). In this way we proceed through the rest of the teeth in $\mathcal{L}_i$, creating edges between teeth where one body is a minimal containing superset of another. This forms something of a comparability tree representing the teeth with a given root. With $\mathcal{L}_i$ sorted by body size, this search can easily be performed by scanning the teeth from largest bodies to smallest.

This procedure of creating an individual branch is illustrated in Figure 5.4, focusing on light, primal candidate teeth from Figure 5.1 with root 4. The label d_4 denotes the degree equation edge, and all other labels refer to the tooth inequality corresponding to that tooth. Note that $(4, \{2\})$ and $(4, \{5\})$ both have $(4, \{2, 5\})$ as a parent, but neither of them is directly comparable to the other. This picture also helps emphasize the correspondence between teeth, inequalities, vertices, and edges. To give an example, the vertex v_2 corresponds to the *tooth* $(4, \{5\})$, but it is the edge between v_2 and its parent v_1 which corresponds to the *simple tooth inequality* for the tooth $(4, \{5\})$. Similarly, the vertex v_1 corresponds to the G^* vertex 4, but it is the edge between v_0 and v_1 which corresponds to the degree equation $x(\delta(4)) = 2$.

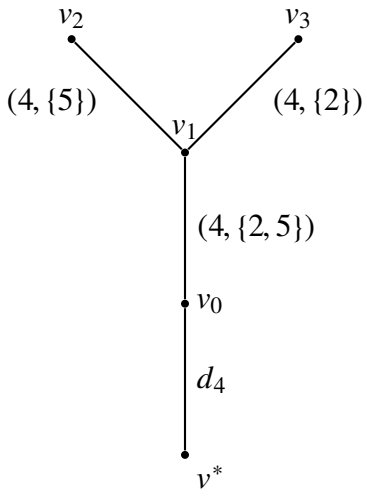


Figure 5.4: The branch of the witness tree representing light teeth with root 4 from Figure 5.1.

To get the overall tree, form the subtrees for the lists $\mathcal{L}_1, \ldots, \mathcal{L}_n$ and join them by identifying the special vertex v^* that is the root of each tree. Note that if a list $\mathcal{L}_k$ is empty, then its subtree is nothing but an edge between v^* and a lone leaf vertex, representing the degree equation for the vertex k.

The procedure described in the previous paragraph does the work of representing the degree equations and simple tooth inequalities. Thus, we still need to handle the nonnegativity inequalities.

The role and use of the nonnegativity inequalities is a bit more subtle. Recall that, for a system $Ax \leqslant b, x \geqslant 0$, a $\{0, \frac{1}{2}\}$-cut takes the form

$$\lfloor \lambda A \rfloor x \leqslant \lfloor \lambda b \rfloor,$$

where λ is a vector with entries in $\{0, \frac{1}{2}\}$ such that λb is not integral. In our particular case, the system $Ax \leqslant b$ describes the linear system of tooth inequalities and degree equations. Thus, if a row of A is multiplied by $\frac{1}{2}$, the corresponding degree equation or simple tooth inequality is *used* in the derivation of the simple DP inequality. However, the nonnegativity inequalities come into play in a different way. Fleischer et al. explain:

> The nonnegativity inequality for a given variable x_j is *used* if the j^{th} coefficient of the vector λA is fractional. (Rounding down the coefficient of x_j on the left-hand side ... is equivalent to adding one half of the nonnegativity inequality $-x_j \leqslant 0$.

Thus, the nonnegativity inequalities are how we encode the operation of rounding down to the nearest integer in the case of a fractional coefficient. With nothing but the tree graph described previously, it would still be the case that cuts could be used to describe simple DP inequalities. But recall that our goal is to be able to make a statement about odd cuts below a certain weight corresponding to *violated* simple DP inequalities.

With this motivation, we consider the Fleischer et al. protocol for adding nonnegativity inequalities to the tree just constructed. Let (i, j) be an edge in E^*. Considering the subtree for $\mathcal{L}_i$, suppose there is a vertex u corresponding to the tooth (i, S) with smallest body S such that $j \in S$. Conversely, consider the subtree for $\mathcal{L}_j$: suppose there is a vertex v corresponding to the tooth (j, R) with smallest body R containing i. In this case, the edge u, v is added to the witness graph, with weight equal to the value $x^*_{i,j}$ from the vector x^* used to define the support graph. We will refer to these nonnegativity edges as *web* edges, and the pair of teeth (i, S) and (j, R) shall be called a *companion* pair defining the web edge.

Figure 5.5 sketches partially the procedure just described: we construct the two separate branches for roots 4 and 5, and join them at the v^* vertex. Moreover, we include the web edge for the support graph edge $(4, 5)$. That is, the tooth $(4, \{5\})$ has the smallest body in the subtree for root 4 containing the vertex 5; and the tooth $(5, \{4, 2\})$ is the smallest (and only) vertex in the subtree for root 5 containing the vertex 4. Thus, $(4, \{5\})$ and $(5, \{4, 2\})$ are the companion pair giving the web edge for $(4, 5)$.

To give an idea of the purpose of web edges, let us consider an extremely simple negative example. In Figure 5.6 we depict a cut in the witness graph where the cut set contains nothing but the vertex for the simple tooth $(4, \{5\})$.

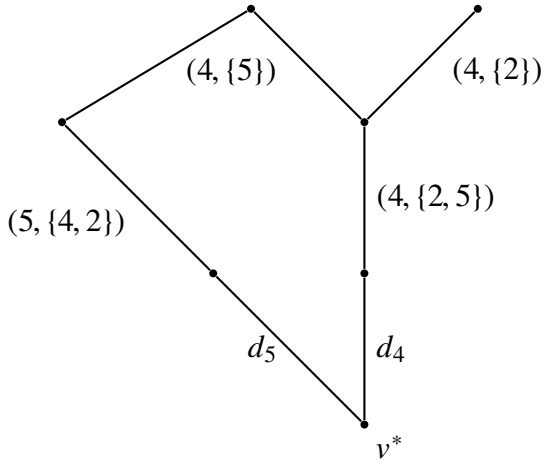


Figure 5.5: The branches for roots 4 and 5 joined at the central node, with a web edge.

Example 5. In reference to Figure 5.6, and the solution vector x^* from Figure 5.1, the cut with vertex set $\{(4, \{5\})\}$ does *not* give a violated simple DP inequality. The edge set of this cut consists of two edges:

- The web edge for the support graph edge $(4, 5)$, having weight $x^*_{4,5} = \frac{2}{3}$; and
- The edge between the tooth $(4, \{5\})$ and its parent $(4, \{2, 5\})$, having weight equal to the slack of the simple tooth inequality for $(4, \{5\})$, which is $\frac{1}{3}$.

Thus, this cut has weight precisely one, indicating that it does *not* give rise to a violated simple DP inequality. And indeed, suppose we were to perform $\{0, \frac{1}{2}\}$-rounding on the simple tooth inequality for $(4, \{5\})$. The tooth inequality itself is just $x^*_{4,5} \leqslant 1$, or $\frac{2}{3} \leqslant 1$. Dividing by two gives $\frac{1}{3} \leqslant \frac{1}{2}$, and rounding down to the nearest integer gives the trivial (and certainly not violated) inequality $0 \leqslant 0$. Note that, without the web edge, the weight of the cut would have just been $\frac{2}{3}$, erroneously indicating a violated simple DP inequality. //

Note that a minor detail is omitted from the instructions for adding web edges. Recall that for an edge (i, j) in E^*, we search the teeth with root i for a smallest body containing j, and search the teeth with root j for a smallest body containing i. But what if no such tooth exists? A tempting answer is that if no such companion tooth exists then we should add no web edge for (i, j). Thanks to Example 5, though, we now know that this would corrupt the structure of the witness graph, causing it to identify simple DP inequalities which are not violated. The answer is as follows. If (i, S) is a simple tooth with $j \in S$, but there are no teeth (j, T) such that $i \in T$,

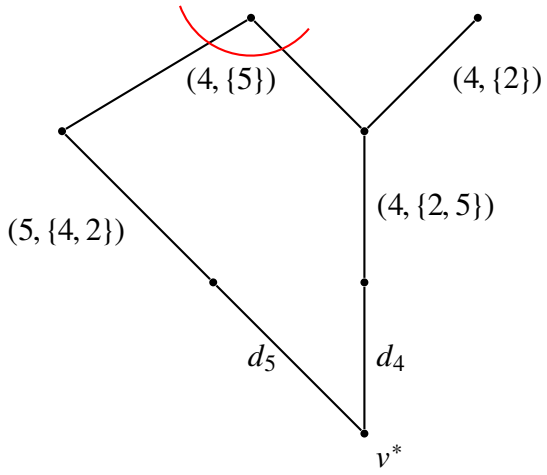


Figure 5.6: A cut in the witness graph with vertex set consisting of a single simple tooth.

then the web edge for (i, j) should go from the vertex (i, S) to the vertex for the literal graph vertex j, i.e., the vertex which would have been called v_0 in the construction of the subtree for root j. Indeed, whether or not (i, S) has a companion tooth, we still need a web edge to encode any integer rounding that would take place if some edge received an odd coefficient in the simple tooth inequality for (i, S).

This is the purpose partially served by the handle nodes, which, recall, are defined as the nodes for which we use the degree equations. A handle set H gives rise to the inequality

$$2x(\gamma(H)) + x(\delta(H)) \leqslant 2|H|,$$

so the edges which occur in $\delta(H)$ give a way to "balance" the coefficients on an edge which receives odd coefficients when considering a simple tooth inequality alone.

Fleischer et al. do not make any mention of what should be done when adding web edges for support graph edges (i, j) such that no companion pair of teeth exists. As we have seen, it is a simple matter to show what should be done, and the authors probably felt that this simple detail could be left as an exercise. I belabour this point only to provide motivation for the promised contraction heuristic that will be developed over the next two sections.

5.3.2 A Contraction Argument

At this point in the chapter, we have described the entirety of the Fleischer et al. separation algorithm, and almost all of the implementation choices made in Camargue. To recap, we find a

collection of candidate teeth, build a witness graph, and then search for odd cuts of weight less than one in the witness graph, each of which gives a violated simple DP inequality. More specific to the implementation in Camargue, we find *light* candidate teeth, create a witness graph, and search for odd cuts in this witness graph. The phase of searching for violated simple DP cuts with a heavy tooth is skipped entirely. The Fleischer et al. algorithm is an exact separation algorithm and, other than the omission of the heavy tooth step, the Camargue implementation described thus far has not made any compromises which may result in missed violated inequalities. We now depart more thoroughly from the exact separation protocol.

The approach taken here is a spiritual successor to the work of Cook, Espinoza, and Goycoolea [7] on standard separation of general (not just simple) domino parity inequalities. Cook et al. base their computational study on the work of Adam Letchford [16], who shows that exact separation of domino parity inequalities is possible in time $O(n^3)$, provided the support graph G^* is planar. Cook et al. describe a computer implementation, developing a host of practical and heuristic enhancements which lead to an optimal solution to the then-as-yet-unsolved TSPLIB [28] instance pla33810. One of the key insights of Cook et al. is the development of a safe shrinking rule with respect to DP inequalities.

Thus, in particular, Cook et al. develop a *domino-safe* shrinking procedure in [7]. We briefly reviewed shrinking rules in Section 4.4.1, but this marks the first attempt at adapting shrinking rules to the primal case. In fact the procedure we describe is *not* a shrinking rule, although the end goal is the same. Moreover, we shall see that it is not safe, in the sense that it may miss some violated simple DP inequalities in exchange for reduced computation time.

The target of this rule is not the support graph G^* itself, but the witness graph arising from a collection of candidate teeth. Computing odd cuts on the witness graph can be a severe computational bottleneck with larger TSP instances, so we want to develop a way to reduce the sizes of the graphs on which odd cuts are computed. The main ingredient is the following extremely obtuse proposition.

Proposition 5.3.1. *Any simple domino-parity inequality identified as an odd cut from some witness graph W can also be identified as an odd cut from a minor of W.*

Proof. Let $\mathcal{L}_i$ denote the list of light teeth with root i, for $i = 1, \ldots, n$, and let $\mathcal{T}_i$ denote the rooted subtree for the list $\mathcal{L}_i$. Thus, W is constructed by joining each of these trees $\mathcal{T}_i$ at the root and adding web edges. Given a simple DP cut derived from an odd cut in W with cut vertices $U = \{u_1, \ldots, u_k\}$, construct a minor W' from W as follows. For every u_ℓ in U, if u_ℓ corresponds to a tooth with root i, then the full tree $\mathcal{T}_i$ appears in W' unchanged. For each j such that no vertex in U corresponds to a tooth with root j, let $\mathcal{T}_j$ be the tree for root j, with v^* denoting the root vertex and v_0 denoting the vertex for the support graph node j, hence (v^*, v_0) is the edge representing the degree equation for j. Then, identify every single vertex in $\mathcal{T}_j$ with v_0, except

for v^*. Thus, we arrive at a tree $\mathcal{T}_j'$ which consists of the root vertex v^* and the contracted vertex $\mathbf{v_0}$. (This is precisely the tree that would be constructed for an empty list of teeth.) As for all the vertices which have been contracted to form $\mathbf{v_0}$, all of their incident edges representing simple tooth inequalities are deleted, and all of their incident web edges are now incident with $\mathbf{v_0}$, with the other endpoint unchanged. Performing this procedure for all cut vertices in U and all subtrees $\mathcal{T}_i$ gives the minor W' of W. The exact same cut set U gives an odd cut of the same weight in W', with the same aggregated inequalities. To see this, consider an edge $e = (u_\ell, w)$ in $\delta(U)$, which either corresponds to a simple tooth inequality, a degree equation, or a web edge. If e is a simple tooth inequality or a degree equation edge, then w is a vertex of the subtree containing u_ℓ, so it was not contracted or modified in any way. If e is a web edge, then w belongs to a subtree distinct from the one containing u_ℓ. Whether or not the tree containing w was contracted, the meaning and weight of the web edge (u_ℓ, w) is unchanged. ∎

This is the type of proposition that gives mathematicians a bad name. It says that, having already identified a simple DP cut from some witness graph, and therefore having undergone the computational effort of computing an odd cut in this witness graph, it would have been possible to obtain the same cut by finding an odd cut in a graph of possibly much smaller size. Having already searched a full-sized witness graph and found a violated cut, it is almost worse than useless to know that the same cut could have been found with less effort. Thus, this proposition informs the motivation, rather than development, of the contraction heuristic employed in Camargue.

Suppose we have a support graph G^* with candidate tooth collection $\mathcal{L}_1, \ldots, \mathcal{L}_n$. The idea is to partition the vertices of G^* into subsets $V_1, \ldots, V_k$ and construct a miniature witness graph W_ℓ for each subset V_ℓ as follows. If W is the full witness graph that would have been constructed, then W_ℓ is obtained by contracting each subtree T_i such that $i \notin V_\ell$, as per the contraction procedure described in Proposition 5.3.1. Thus, effectively, we just ignore all lists of teeth with roots not in V_ℓ. The challenge is to choose a partition such that

- each subset V_ℓ has relatively small size, so it is easy to compute an odd cut on W_ℓ; and
- a small number of cuts are lost as compared to the number that would have been found by searching the full witness graph.

The next section describes a possible partition choice, along with computational results.

5.4 Computing with Contracted Witnesses

This chapter concludes with computational tests of the simple DP inequality separation procedure implemented in Camargue. We will describe a heuristic for choosing vertex partitions, evaluating

its performance. The performance benchmarks are guided by very simple principles: we report computation time and number of cuts returned as compared to the implementation which uses no partition for the witness graph.

All test cases are performed as follows. Concorde is used to compute an optimal subtour polytope solution for a given TSP instance, along with a starting chained Lin-Kernighan tour. The edges of the tour and LP solution are then loaded into Camargue, which is used to find a collection of light, primal simple candidate teeth. At this point the approach diverges based on partition choice (or lack thereof), computing one or more witness graphs on which to search for cuts.

5.4.1 Geometric Partitions

We consider an approach that is suitable for application on geometric or Euclidean-distance TSP instances. It is based on the work of Karp [15], concerning partitioning algorithms as a general solution protocol for the TSP, rather than as a cut separation procedure. Given a TSP instance, Karp fixes some desired maximum partition size, repeatedly subdividing the horizontal and vertical coordinates of the instance to create sub-rectangles enclosing the vertex subsets. The idea is to make the partitions so small that optimal tours on these vertex subsets can be computed quickly, perhaps even by exhaustive methods. Then, the optimal subtours are stitched together in some way to provide a tour for the original instance. Thus, all we are borrowing is the idea of creating geometric subdivisions of the vertex set with some fixed maximum partition size.

Figure 5.7 should provide some intuition for Karp partitions, and more generally for the structure of simple DP inequalities. It is generated using the test protocol just described, with no partitions or contracted witness graphs. For every violated, primal simple DP cut found, Figure 5.7 highlights each edge that is assigned a nonzero coefficient by at least one cut. (Purely for the sake of creating an appealing illustration, we ignore some higher density cuts.) Overlaid atop these nodes and highlighted edges are the bounding rectangles for a Karp partition of the vertex set.

A lot of rich structure is evident in Figure 5.7. At the beginning of this chapter we remarked that simple DP inequalities are a proper generalization of the simple comb inequalities, and indeed many simple combs are evident. A total of 23 cuts were used to generate the edge sets in Figure 5.7, but 19 connected components are shown. The larger connected components in the top and bottom halves of the illustration therefore represent some overlap between a handful of cuts. So while we should not conclude that a simple DP inequality may look exactly like either of these larger two components, the illustration still hints at the complex structure these cuts may assume, especially as compared to the simple comb inequalities, or blossoms.

More relevant to the topic at hand, Figure 5.7 paints a sunny picture of Karp partitions as

an approach for contracting witness graphs. To be sure, certain pairs of partitions, such as $4, 6$ and $9, 11$, appear to contain edges from cuts that do not fit neatly into just one bounding box. Most every other cut fits nicely within a single region, though: there is just a single edge overlap evident between boxes 13 and 15. Of course it should be noted that pr2392 is a circuit-board drilling TSP instance, and as such it already exhibits a structure of neatly partitioned vertices.

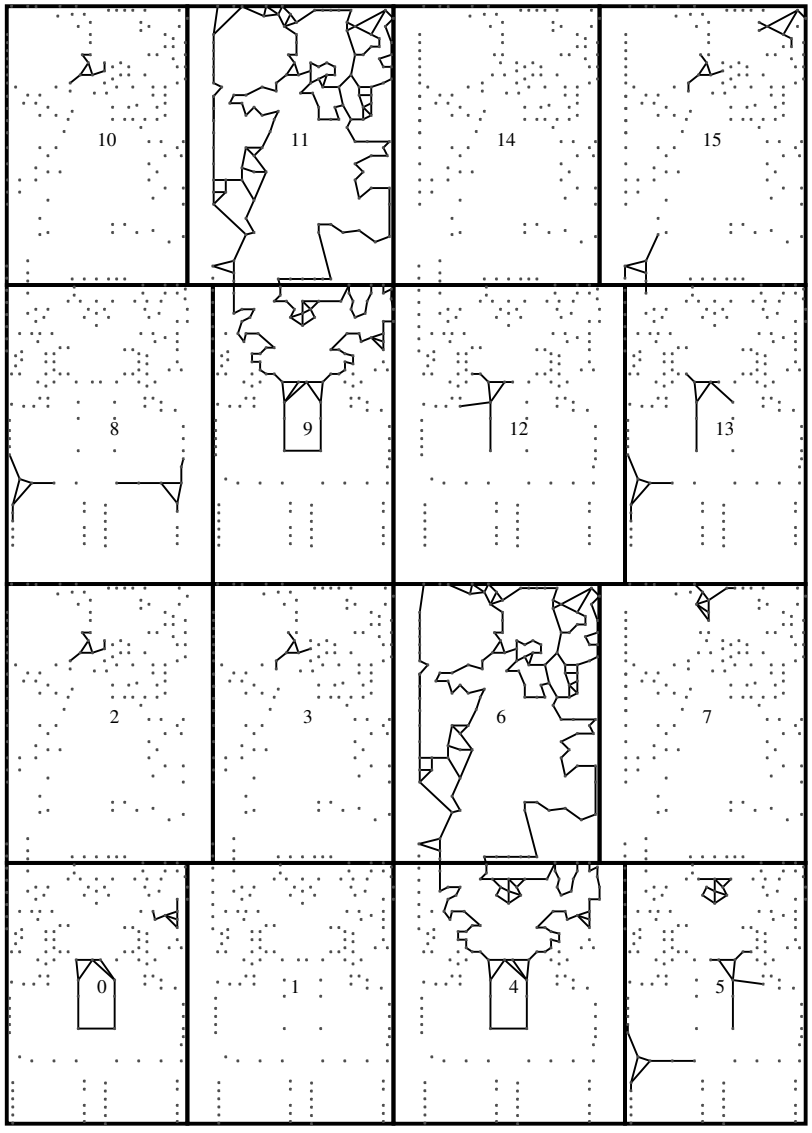


Figure 5.7: Edges from violated primal simple DP inequalities in an optimal subtour polytope solution vector for pr2392, with Karp partition boundaries numbered and overlaid.

We also consider the same protocol applied to the TSPLIB instance dsj1000, which is described as a "clustered random problem". As such, we might expect the instance to be less amenable to Karp partitioning, a suspicion confirmed visually in Figure 5.8. Still, the illustration also shows that a fair number of cuts lie properly within the established bounding boxes.

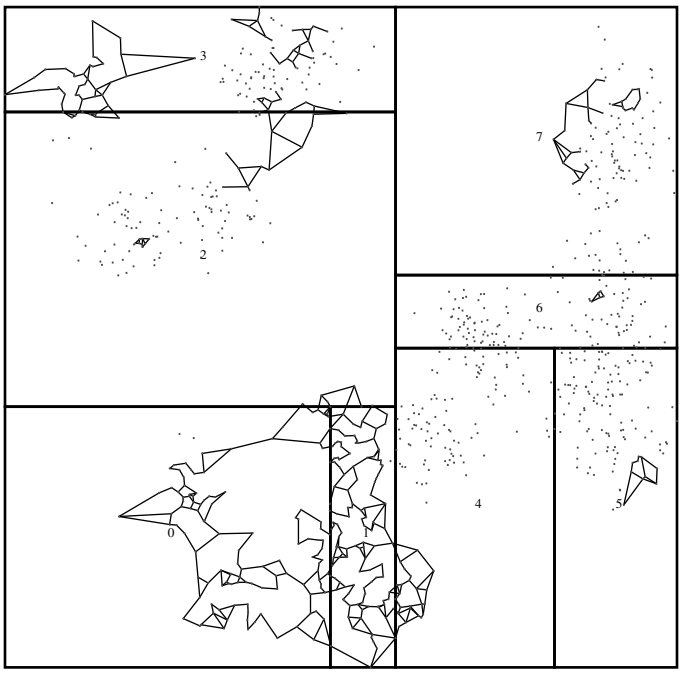


Figure 5.8: Edges from violated primal simple DP inequalities for dsj1000, with Karp partition boundaries numbered and overlaid.

At this point the use of Karp partitions has been sufficiently debated and motivated: some computational experiments are in order. As described at the beginning of this section, test cases are constructed with tours and subtour polytope optima computed by Concorde, with simple DP inequalities separated by Camargue. Karp partitions are chosen to have a maximum subset size of $4 \cdot \sqrt{n}$, where n is the instance node count. The test bed consists of all TSPLIB [28] instances of size greater than 3,000 nodes, excluding the largest, pla85900.

The raw test data appears in Table 5.1. Tests were performed on a 3.5 GHz Intel Xeon server. Since the test computer is a shared resource, and to dull the impact of unusually fast or slow runs, five separate trials were conducted, with identical random seeds used for all non-deterministic subroutines. Thus, the column "DP cuts found" indicates the number of violated, primal simple DP cuts found, which is the exact same every time. The column "CPU seconds" indicates the geometric mean of CPU times for each trial. For each instance, these results are divided into rows

by “No Partition”, indicating no vertex partition, and “Partitioned”, indicating the Karp partition procedure just described.

Table 5.1: CPU times and number of cuts found for partitioned and unpartitioned simple DP separation.

Instance		CPU seconds	DP cuts found
pcb3038	No Partition	1.73539	100
	Partitioned	0.258309	98
fl3795	No Partition	2.96844	31
	Partitioned	0.517282	29
fnl4461	No Partition	3.51143	167
	Partitioned	0.497919	137
rl5915	No Partition	7.1301	116
	Partitioned	0.926458	116
rl5934	No Partition	7.2053	100
	Partitioned	0.867667	98
pla7397	No Partition	9.7413	93
	Partitioned	1.35257	90
rl11849	No Partition	28.7045	266
	Partitioned	2.86735	241
usa13509	No Partition	37.1939	446
	Partitioned	3.11273	417
brd14051	No Partition	39.0455	435
	Partitioned	3.14132	395
d15112	No Partition	46.145	493
	Partitioned	3.31969	462
d18512	No Partition	75.2051	605
	Partitioned	5.8963	553
pla33810	No Partition	351.254	511
	Partitioned	19.1909	457

The results from Table 5.1 are reproduced graphically in Figure 5.9. The bars, with scale on the left vertical axis, indicate the ratio of CPU time for unpartitioned separation vs separation with partitions and contractions, thereby showing the speedup factor obtained by using the Karp partitions. The scatter points, with scale on the right vertical axis, indicate the number of cuts returned by partitioned separation as a fraction of the number of cuts returned for unpartitioned separation.

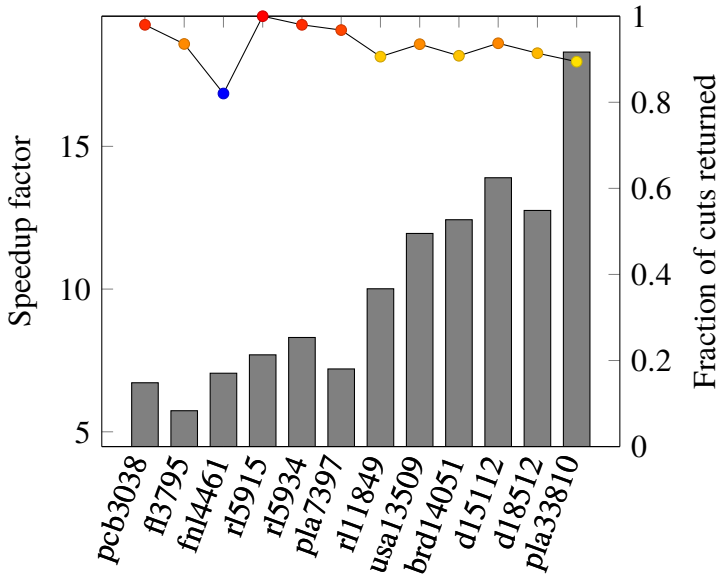


Figure 5.9: Bar graph of scaled CPU times.

The results from Table 5.1 and Figure 5.9 show clearly that Karp partitions provide an effective way to speed up simple DP separation without missing too many violated cuts. Every single instance showed a speedup factor of at least five, and this speedup seems to become more pronounced when considering larger instances. Cut retention seems favourable too, with the worst case being fnl4461, where only 82% of cuts were found in the partitioned instance. But in all other cases at least 89% of the cuts were found, with rl5915 matching the number of cuts found in the unpartitioned case.

A nice feature of the partition approach to simple DP computation is the opportunity to search over contracted witness graphs in parallel. Indeed, odd cuts on different contracted witness graphs are completely independent of one and other. The tests reported were carried out with a serial implementation, but Camargue is configured to allow a parallel implementation to be used during the actual primal cutting plane solution process. Some simple arithmetic shows that this result is less exciting than it may seem at first glance though. With a maximum partition size of $4 \cdot \sqrt{n}$, the number of partitions is evidently at most $\sqrt{n}/4$. For instances even as large as 10–15 thousand nodes, this may result in a total number of partitions which is so low that the net speedup obtained is relatively minor, given the overhead of managing a parallel search.

Case Study: pla85900

We end this section by indulging in a more focused case study of the TSPLIB instance pla85900, which was omitted without explanation from the results in Table 5.1. Simply put, the test run for pla85900 was terminated when unpartitioned simple DP separation remained unfinished after over twenty minutes. Thus, anecdotally, simple DP separation more or less will not run on such a large instance without any contraction procedure applied. Rather than compare the effect of partitioning or not partitioning, we will attempt to profile the separation process with a Karp partition applied. Moreover, we will try to quantify the effect of parallelizing the search for simple DP cuts, a feature absent from the other experiments in this chapter. Tests in this section were performed on a 4-core 3.5 GHz Intel Xeon server, with wall clock times reported rather than CPU seconds, and a single trial done for each individual test case. The aim of this section is more to qualitatively profile the code, rather than establishing hard running time bounds/factors.

In Table 5.2, we report running times of subroutines for the serial implementation of candidate tooth generation and simple DP separation. The row "Candidate Generation (overall)" is the sum of all the rows above it, the total time required for generating the collection of candidate teeth. "Sorting/Allocating" refers to the time spent running Algorithm 5.2.1. "Finding Teeth" refers to invoking Algorithm 5.2.3 within ALL-SEGMENTS. "Sort by Root" indicates the amount of time sorting by body size each tooth list. Below that, with the collection of candidate teeth in hand, the row "Witness Computation" refers to the time spent building the contracted witness graphs, building the Gomory–Hu trees for these graphs, finding odd cuts, and translating them into simple DP cuts. Finally, "Cuts Found" is the number of violated primal simple DP cuts found.

Table 5.2: Profiling serial, partitioned simple DP cut separation for pla85900.

Subroutine	Wall Clock Time (s)
Sorting/Allocating	0.051650
Finding Teeth	0.557567
Sorting by Root	0.003643
Candidate Generation (overall)	0.561214
Witness Computation	197.570756
Cuts Found	713

It comes as no surprise that building the witnesses and computing odd cuts is the most intensive computational task. Perhaps the most remarkable feature, though, is the speed of the candidate tooth generation process: even on an instance of this size it is almost negligible in the overall running time of the routine. This is due in large part to the elegant implementation of ALL-SEGMENTS provided by Applegate et al. Another important factor, though, is the storage

overhead saved by eliminating in place. Recall that Letchford and Lodi [20] warned of having to store a collection of teeth having size $O(n^3m)$, and then applying elimination steps to this list. Taking the edges of the tour and LP solution generated by Concorde, the value of m in this test was 89,066. To give an idea, storing n^3m 4-byte integers would require around a quarter of a million *petabytes* of storage space.

In Table 5.3, we report times for all the same subroutines using a parallel implementation. Explicitly, the outer **for** loop of Algorithm 5.2.1 is run in parallel, and we separate over Karp partitioned witness graphs concurrently. The actual search for candidate teeth is implemented exactly as before.

Table 5.3: Profiling parallel, partitioned simple DP cut separation for pla85900.

Subroutine	Wall Clock Time (s)
Sorting/Allocating	0.062050
Finding Teeth	0.251401
Sorting by Root	0.005928
Candidate Generation (overall)	0.319397
Witness Computation	59.203647
Cuts Found	713

In Table 5.3 we see that "Finding Teeth" was twice as fast as in Table 5.2 (despite being implemented identically) which just gives an indication of the variance that can take place between runs. Generally speaking, the effect of parallelizing the candidate tooth phase is pretty negligible. "Witness Computation" experiences a nice speedup, running about three times faster. It seems that for this phase of separation, a parallel implementation provides a good practical performance boost.

A final experiment is reported in Table 5.4. All the time measurements from previous trials are reproduced, but the only one of interest is the last line, "Witness Computation". This test was implemented exactly the same as in Table 5.3, but with a short circuiting step in the witness graph search. Choosing an arbitrary target, the Camargue implementation was set to terminate the separation procedure after at least 500 cuts had been placed in a queue for addition, constructing no further witness graphs afterwards.

Here we get another nice speedup: "Witness Computation" is about twice as fast as in Table 5.3; but also the number of cuts returned is about half as much. This deserves a remark: Letchford and Lodi's characterization of primal candidate teeth is necessary but *not* sufficient for an inequality to be tight at the current tour. Thus, separation was terminated with a queue of 644 cuts to be examined, and 393 of them turned out to satisfy the primal separation problem.

Table 5.4: Profiling parallel, partitioned simple DP cut separation for pla85900, with a cutoff for cut generation.

Subroutine	Wall Clock Time (s)
Sorting/Allocating	0.085429
Finding Teeth	0.644775
Sorting by Root	0.010383
Candidate Generation (overall)	0.740625
Witness Computation	32.515061
Cuts Found	393

When embedded into the actual primal cutting plane solution process in Camargue, simple DP separation is configured as in Table 5.4, with 250 cuts set as the target for short-circuiting the computation.

Note that this same short circuiting procedure could have been implemented with the serial implementation of Table 5.2, but an implementation is *not* readily apparent without using some sort of partitioning scheme. My preliminary experiments showed that the Gomory-Hu tree computation accounts for almost all of the time in the witness graph phase of the algorithm. So if no partitions are used, there would be nothing to gain from computing an exceptionally large Gomory-Hu tree whose traversal is terminated at some arbitrary point. Rather, the advantage of the partition approach from a practical point of view is that we can stop the computation *before* any new witness graphs or Gomory-Hu trees need to be constructed.

Chapter 6

Managing the Linear Programming Problems

In this chapter, we are concerned with the question of how to manage the linear programming problems that arise in primal cutting plane TSP computation. The title of this chapter is borrowed exactly from Chapter 12 of [2], in which Applegate et al. treat the same questions from the point of view of their Concorde TSP solver. The concepts covered here are thus mostly a proper subset of those from Chapter 12 of Applegate et al. The idea is not to rehash descriptions of their algorithms and data structures. Rather, we want to know how and when certain approaches should be applied, given various points of divergence between the primal and dual fractional approach.

In this chapter we will speak of the core LP relaxation (or just core LP) currently under consideration for the TSP instance at hand. The question is how to keep a good sample of edges and cuts in the core LP so as to augment or certify the optimality of tours, while avoiding overly long computation times for non-degenerate pivots. We shall first lay some simple preliminaries, commenting on straightforward adaptations from the dual fractional case to the primal case. Then we discuss cut management heuristics and edge pricing. The chapter concludes with a discussion on the possibility of fixing variables by reduced cost.

The key point of tension is the presence or absence of optimal primal and dual LP solutions, and the way subsequent solutions are related to each other. In Concorde, heuristics for managing cuts and edges are based on moving through successive dual optimal solutions, each of which provides a starting basis for the following one. But in the primal case, we move from tour to tour, again reusing starting bases. This points to the common theme we try to extract: as we move from solution to solution, what can the current one tell us about the next one?

6.1 Preliminaries

To fix the discussion for the rest of the chapter, let us name explicitly the approaches that will be adopted from Chapter 12 of Applegate et al. [2] with virtually no adjustments. These are as follows.

- *The initial edge set:* we use the union of edge sets of 10 tours obtained by short runs of the chained Lin-Kernighan algorithm (see Section 12.1.2 of [2]), while also explicitly adding the edges of the best starting tour $\bar{x}$.
- *The representation of cuts:* outside of the LP solver, we represent cuts in a vertex-centric *hypergraph* form, as in Section 5.8 of [2], and reviewed below.
- *The compressed representation of cuts:* we represent the vertex subsets of a hypergraph as a union of segment intervals from the tour $\bar{x}$, as in Section 12.2.2 of [2], also reviewed below.
- *Edge pricing scans:* when pricing over the edge set, we price edges in batches by computing reduced cost estimates, limiting the number of edges for which reduced costs need to be computed explicitly (see Section 12.3 of [2]).

As for the cutting planes themselves, the collection of cuts is partitioned in two. The active cuts are those currently in the core LP. They are represented internally by the LP solver, and mirrored exactly in a compressed hypergraph representation which allows pricing edges which are not in the core LP. The inactive cuts are kept in a *cut pool*, also in the compressed hypergraph representation.

From the points enumerated above, let us remark on the compression technique used for hypergraphs. To use the notation of Applegate et al., a hypergraph consists of a pair $\mathcal{H} = (V, \mathcal{F})$, where V is the vertex set of the TSP instance, and $\mathcal{F}$, the *edges* of $\mathcal{H}$, consists of subsets of V. The left-hand side of a hypergraph inequality at a given vector x is

$$H \circ x = \sum (x(\delta(S)) : S \in \mathcal{F}),$$

with the right-hand side dependent on the template of cut being represented. For example if $\mathcal{F} = \{S\}$, the hypergraph $(V, \mathcal{F})$ is the subtour inequality $H \circ x \geqslant 2$. And if $\mathcal{F} = \{H, T_1, \ldots, T_k\}$ are the handle and tooth sets of a comb, then $H \circ x \geqslant 3k + 1$ is the comb inequality with handle and tooth set $\mathcal{F}$.

In Concorde, Applegate et al. fix some starting tour $\bar{x}$ which gives an upper bound, and is also used to search for segment cuts. We described explicitly in Section 4.1.1 how primal separation

of SECs is used by Applegate et al. as a fast heuristic for standard subtour separation. With the same logic they use to justify this heuristic, Applegate et al. propose to try to represent the edges of a hypergraph using intervals from $\bar{x}$ as well, writing that "sets from our cutting planes can be expected to consist of a small number of intervals from an optimal tour." They verify this empirically as well in Section 12.2 of [2].

In the primal cutting plane case, and in Camargue, we use the same compression technique, but it comes with somewhat of a stronger guarantee. Indeed in Chapters 4 and 5, we saw that tour intervals play a non-trivial role in the separation of *all* template cuts under consideration. On the other hand, Applegate et al. fix their starting tour $\bar{x}$, using it to generate segment cuts even if other tours are encountered later in the solution process. But in the primal case, this tour $\bar{x}$ may be updated a number of times throughout the solution process, say to some tour x'. From then on, x' is used as the source of cuts. But should we use x' to represent the vertex subsets of these new cuts too? This would provide a greater compression factor for new cuts found, but incurs the expense of storing the tour and permutation vector for each of the previous tours, or of computing a new representation for every cut currently stored. In Camargue I have opted just to fix the original $\bar{x}$ as the reference tour for all cuts found regardless of augmentation.

6.2 Cut Storage and Pools

We first consider the question of adding and removing cuts from the core LP, and the management of a cut pool. In Section 12.1.5 of [2], Applegate et al. describe some of the practical measures they use to manage cuts in Concorde. Given an LP solution x^*, and the cuts pre-existing in the core LP which led us to this x^*, they consider three main metrics which determine whether a cut should remain in the core LP. These are

- *Tightness:* does the cut hold as an equation at x^*?
- *Basic status:* is the slack variable corresponding to a cut in the optimal basis for x^*?
- *Dual values:* in the dual solution for x^*, is the dual variable for a cut above some threshold?

Thus, all these metrics are used to determine whether a cut should remain in the core LP; and if not, should the cut be stored in a cut pool for later use. Briefly, they make the following decisions.

- *Tightness:* cuts which have become slack at x^* are moved to the cut pool.
- *Basic status:* if the slack for a newly added cut is in the basis for x^*, it is deleted from the core LP immediately, without adding it to the pool.

– *Dual values:* if a fixed number of new solutions x^* have been generated in which the dual value for a cut is below some positive threshold, the cut is moved to the pool.

Let us adopt these three metrics in the primal case as well, but with some adjustments. We fix the current best tour $\bar{x}$, its primal feasible basis, and a non-tour pivot x^* obtained from $\bar{x}$.

First and simplest, we consider the primal interpretation of tightness as a metric. Well, in the primal case *all* our cuts, with the possible exception of connected component SECs discussed in Section 4.1.2, are tight at $\bar{x}$. In [18] Letchford and Lodi write the following.

> Each time an augmentation (i.e., an improvement) occurs . . . cuts which have become slack can be moved from the LP to the pool and cuts which have become tight can be moved from the pool to the LP.

Thus, we (mostly) only find tight cuts as the solution progresses, and slack cuts are removed with each augmentation. Note that any slack connected component cuts would be moved to the pool at such an augmentation as well.

Next we adapt the metric of basic status. This requires a bit of thought, but it is still straightforward. Given that our goal is to find an optimal basis for the tour $\bar{x}$, we should dispose of cuts whose slack variables are in the latest primal feasible basis found for $\bar{x}$.

The metric of dual values provides the most ambiguity. First, should we be interested in the dual values at $\bar{x}$, or x^*, or possibly both? In the dual fractional case, this heuristic is easily justified. As we move from lower bound to lower bound, a non-tour solution is separated by cutting planes, with its starting basis used for a subsequent call to dual simplex optimization. But at this point it is helpful to recall the discussion of Section 3.3.4, in which we compared the nature of subsequent bases in the primal and dual case. The key takeaway from that discussion was that, in the primal case, we expect that subsequent non-degenerate pivots will have very different bases, given that primal cutting planes redirect optimization at each step. Again, Figures 3.6 and 3.7 may provide some helpful intuition.

Applegate et al. write in Section 12.1.5 of [2] that

> [if] for L_2 consecutive LP solves the dual variable is less than some fixed tolerance ε_2, then the cut is deleted from the LP. (Typical values are $L_2 = 10$ and $\varepsilon_2 = 0.001$.)

One approach, centred around non-tour pivots, would be to apply this metric with a higher, and thus more conservative, value of L_2. Non-tour pivots may well have dissimilar bases and dual solutions. But if we compute a larger number of consecutive pivots in which a cut assumes a small dual value, this may be a hint that it is not actively guiding the solution process. But Applegate et al. also describe the specific motivation for this heuristic:

> This deletion condition is weaker than the standard practice of deleting cuts only if they are slack (that is, only if they are not satisfied as an equation by the current x^*); for our separation routines on large TSP instances, we found that the core LP would accumulate a great number of cuts that were satisfied as an equation by the current x^* if we only removed slack inequalities.

In primal TSP computation, it seems to be the case as well that deleting slack cuts upon augmentation still causes the core LP to accumulate a great number of constraints. The starting tour may be optimal, in which case no such deletion will take place; and it also sometimes happens that successive tours are so similar in structure that few cuts become slack, if any at all. Thus, it seems reasonable to apply the dual variable metric to tours as well. Fixing the threshold ε below which a dual variable is considered small, we define separately the notion of *tour age* and *pivot age* of a cut. By monitoring the dual variables at each primal pivot, and at each new basis for $\bar{x}$, we can increment the respective tour and pivot ages of a cut, resetting an age to zero if the dual variable exceeds ε. Thus, we define separate constants L_P for pivots and L_T for tours. If a cut's tour age exceeds L_T, or if its pivot age exceeds L_P, it is removed from the core LP. At an augmentation, or if a tour is optimal for its current edge set, the ages of all cuts are reset. Camargue opts for fairly conservative values of L_P and L_T, using $L_P = 30$ for pivots and $L_T = 20$ for tours.

6.3 Edge Pricing

The next topic at hand is the question of pricing edges for inclusion in the core LP. The solution process begins with a sparse edge set, and we do not propose to alter in any serious way the *method* of scanning for edges with negative reduced cost that should be considered for addition to the core LP. Rather, the issue is how frequently, and under what circumstances, should such a scan be performed. We briefly sketch the approach taken by Applegate et al. in Concorde, before describing alterations made in Camargue.

In Section 12.3 and 12.4 of [2], Applegate et al. describe an edge pricing scan which works in two modes: one cycles through the 50-nearest neighbours edge set, and the other scans through the complete graph edge set. The authors describe a control program which is responsible for alternating between cut separation and edge pricing, using an evaluation function on the progress of the dual lower bound obtained by cuts that have been added. Essentially, they remark that when "the bound looks too good to be true, then it may well be due to an insufficient collection of edges in the core LP." Thus, a key aspect of the edge pricing machinery of Applegate et al. is to re-optimize the core LP after adding a batch of edges, in this way seeing whether "the gains in the LP value disappear after an edge-pricing step."

We propose in the primal case to operate edge pricing in two modes as well; these modes are invoked by analogous measures of progress, and of what it means for a result to look too good to be true. Whenever an augmenting pivot occurs, we perform what might be called a cursory scan of the 50-nearest edge set. That is, we cycle through the edge set, searching for edges that may have negative reduced cost. But if any are found, a relatively small batch of at most some fixed size (Camargue, like Concorde, uses 100) is immediately added to the core, without any further pivoting or optimizing. This choice is guided by the idea that, with every augmentation that occurs, we reach closer to the useful limit of the current edge set. That is, we may be approaching a point where the current tour is optimal for the core edge set.

This brings us to the event that is used to trigger a complete graph pricing scan. Namely, we scan the full edge set whenever the LP solver reports that the current tour is optimal. This forms a nice analogy with what Applegate et al. refer to as a dual lower bound that looks too good to be true. Recall that our initial edge sets consist of the union of 10 chained Lin-Kernighan tours, together with the edges of a tour also likely found by some variation of chained Lin-Kernighan. This procedure gives a nice sampling of edges, but the resulting graphs may be quite sparse, often with many edges having degree exactly two, so that the solver may report on many separate occasions to have certified an optimal tour. Thus, as with dual bounds, we arrive at a notion of a solution looking too good to be true.

In the pricing scan for seemingly optimal tours, we do a thorough edge pricing scan of the complete graph edges that more closely resembles the approach of Applegate et al. We cycle through the edges, computing reduced cost overestimates for a relatively large candidate batch of edges (around 20,000). We then compute genuine reduced costs for a fixed number (around 1,000) of these edges at a time, adding them to the core LP if some are indeed found to have negative reduced cost. Once the edges are added to the core, we primal optimize the LP again, recomputing reduced costs for the remaining edges in the queue. A tour is genuinely optimal if, after scanning through all the edges, adding negative reduced cost edges, and primal optimizing, if after all that the optimal objective value is still close enough to the tour to certify its optimality. Of course, it may also happen that adding a batch of edges drives down the optimal objective value for the current edge set, showing that more work needs to be done to augment or certify the optimality of the current tour.

6.4 Variable Fixing

In our description of full edge pricing, we mentioned quite casually the act of optimizing the LP, perhaps even repeatedly, with the primal simplex algorithm. In all our discussions and implementations thus far, the act of optimization has been completely absent, save for comparison

to the dual fractional case. This point is more significant than it might seem, as it admits the possibility of fixing variables based on reduced cost.

Both cut management and edge pricing were addressed by Letchford and Lodi in their work on primal cutting plane methods [18]. We quoted directly their remarks on cut pools, and on the matter of edge pricing they say simply that it "can also be done easily", which indeed we have just seen. On the matter of variable fixing, however, they are less optimistic, writing that

> . . . fixing variables on the basis of reduced costs seems more problematic. This is because the primal approach as stated yields a sequence of [upper] bounds but not [lower] bounds.

Note that in [18], Letchford and Lodi's prototypical optimization problem is one of *maximization*. Thus, for our purposes, occurrences of "lower" and "upper" bounds are swapped in this discussion: primal TSP computation yields with each tour a new upper bound on the length of the shortest tour, but it does not produce lower bounds. This concern is well-founded, but the authors take a more practical point of view, going on to write the following.

> Of course, if for some reason the code has to be terminated before optimality has been proven, we can simply do what Padberg and Hong did, and solve the current LP to optimality to obtain [a lower] bound. This LP can be solved in a relatively small number of primal simplex pivots, because $\bar{x}$ can be used as a starting basis.

Realistically, nothing (other than a commitment to purity of method) prevents us from periodically primal optimizing the core LP, rather than just performing primal pivots. In moderation, the net computation time incurred by such calls would be low, and it would provide useful information from the point of view of edge elimination, especially if the core edge set is becoming unwieldy. But with the edge pricing protocol used in Camargue, a slightly more natural approach is possible.

We return to the act of optimizing the core LP during complete edge pricing scans. We enter such a scan with $\bar{x}$ as the resident LP solution, along with an optimal basis for $\bar{x}$. If we want to know whether new edges added to the LP have rendered $\bar{x}$ suboptimal, we could just as well see if $\bar{x}$ admits a distinct non-degenerate primal pivot. But as Letchford and Lodi wrote of Padberg and Hong's approach, we can also just optimize the LP with the primal simplex algorithm, using the basis associated to $\bar{x}$ which is a fortiori dual feasible in this situation. And of course a strength of the primal simplex algorithm is that the bases it produces are robust under the addition of edges, providing a useful starting basis after columns have been added. Thus, one could hardly begrudge our decision to optimize here, rather than pivoting.

Recall again the possible outcomes from a complete pricing scan. Given a tour that looks optimal, it is indeed optimal if no edges with negative reduced cost are found, or if some are

added but the optimal objective value remains close enough to the tour. More explicitly, say $\bar{x}$ has length $c^T\bar{x}$. Since edge costs are assumed integral, a tour that strictly improves on $\bar{x}$ cannot have length greater than $c^T\bar{x} - 1$. Thus, if z^* is the optimal objective value obtained after a full pricing scan, then $\bar{x}$ is certified optimal if $z^* > c^T\bar{x} - 1$. In the other possible outcome, adding edges brings us to a new optimal LP solution x^* such that $c^T x^* \leqslant c^T\bar{x} - 1$. The upshot of this outcome is that we now have an optimal primal-dual solution pair (x^*, π). As in Section 5.4 of [2], we can use π to compute an exact valid lower bound in fixed-point arithmetic, similarly computing exact reduced costs for all the edges. This exact lower bound, together with the tour upper bound and reduced costs, allows us to remove edges from the core LP, or fix them to their upper bound of one. In Camargue this seems to produce a nice push-pull relationship between edge pricing and edge elimination: if a pricing scan adds a vast number of negative reduced cost edges to the core LP, effecting a decreased lower bound, this addition of edges may be offset somewhat by reduced cost elimination.

Again, for a less conservative approach, or if the tour is never certified optimal, we could just optimize the core LP at chosen points in the solution process. This approach of periodic optimizing and fixing is not adopted in Camargue. Instead, as described by Applegate et al. in Section 14.2 of [2], manual optimization and edge fixing only takes place at the root node of a branch and cut search, a matter we discuss in the next chapter.

6.5 Cut Separation Control

Throughout this chapter we have sketched most aspects of the core LP *control program*, or *controller*. The controller is responsible for managing the addition and deletion of cuts to the core LP, but one aspect of its operation remains unspecified. Namely, in the task of interleaving cut and column generation, we need some way to determine appropriate circumstances for proceeding through the separation routines described in Chapters 4 and 5, and for separation over the cut pool described in Section 6.1.

Cut separation routines are ordered based on estimates of computational difficulty. We start with cut pool separation and SEC routines, before moving on to heuristics for fast blossoms and block combs. Next is exact blossom separation and simple DP separation, the most demanding of the primal separation routines considered. Section 4.4 mentioned using some more involved standard heuristics for cut metamorphoses and local cuts; these are used after primal exact blossoms and simple DP cuts. Finally, if applicable, is primal separation of safe Gomory cuts.

The loop is broken off by tour pivots, or when all separation routines fail. If a routine fails to return cuts, then certainly the next one should be tried. But in other situations when cuts are found, when should we prefer advancing in the loop to restarting it? And when is the progress

obtained by a full round of separation routines deemed insufficient?

In the previous section we alluded to the approach taken by Applegate et al. in Concorde, described in Section 12.4.3 of [2]. To review, they use an evaluation function which measures the gap being closed between the dual lower bound and the length τ of the best known tour. That is, fixing a given edge set and an initial dual lower bound β, the addition of cuts should result in a new lower bound β' that strictly improves on β, approaching the value τ. Thus, based on the value $\tau - \beta$, Applegate et al. set target values for $\tau - \beta'$ which give a notion of good progress, "too good" progress which suggests the need for column generation, and a modest amount of progress which suggests that a new separation routine should be tried.

We propose to adopt a similar measure in the primal case, but the approach is not so straightforward. Simply put, the objective values of non-degenerate pivots do not follow a strictly increasing progression. To give an example, Figure 6.1 shows the sequence of objective values obtained by a run of Camargue on the TSPLIB [28] instance swiss42, a small example which is solved exactly in a total of 18 non-degenerate pivot steps without any cut metamorphoses, local cuts, or Gomory cuts. The initial optimal tour of length 1273, indicated as Pivot Number 0, is the starting tour, used as a starting basis for subsequent pivots. Note several separate points where the objective value reaches the upper limit of the starting tour, indicating that the tour appeared optimal for the edge set, leading to a round of edge pricing. On the other hand, there are also several pairs of consecutive pivots with no change in objective value; but then these were followed shortly after by a pivot giving an optimal basis for the tour. This hints at the difficulties that can arise by applying a measure of progress that is too coarse, or too greedy.

This perhaps provides a good idea of what it means for primal cutting planes to redirect the course of the simplex algorithm at each step. Rather than an increasing sequence of objective values, we obtain a sequence which varies erratically, with values taking on a more restricted range just before a tour becomes optimal for the current edge set. This guides our notion of progress for primal cutting plane generation: a round of cuts looks good if it causes a non-degenerate pivot vector which differs a great deal from the previous one encountered. Note that here, we are implicitly following the discussion of Section 3.3.1, using objective value as a means to measure the difference between two subsequent pivots.

Again, objective value is not a perfect way to differentiate between solution vectors, although some remedial safeguards are possible. For example, in the context of connected component SEC generation, symmetries in a node set can mean that a subtour eliminated from one geometric region will "reappear" almost identically elsewhere. Thus in Camargue, rounds of SECs that take us from disconnected pivots to connected ones are always regarded as significant, no matter the change in objective value as they are generated.

So loosely speaking, the metric we have settled on is measuring how "different" subsequent pivots are after adding a round of cuts. What is a good way to quantify this? Suppose again that

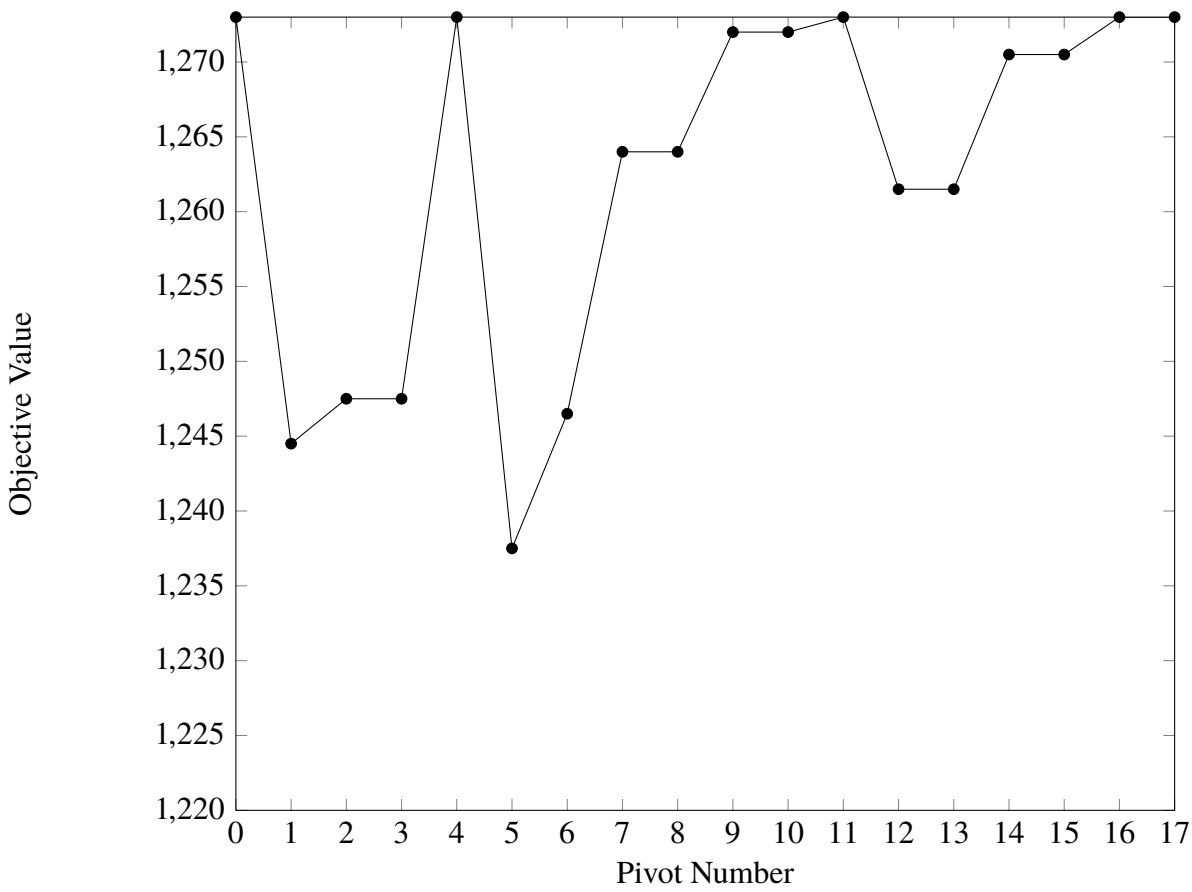


Figure 6.1: A sequence of pivot objective values for swiss42.

$\bar{x}$ is the tour, and let τ denote the length of $\bar{x}$. Let x^* be a pivot of objective value β_1, and let x^{**} be a pivot of value β_2. Suppose, moreover, that x^{**} is obtained from $\bar{x}$ by adding cutting planes which separate x^*, pivoting back to $\bar{x}$, and pivoting again. A few things seem clear:

- we should be interested in the difference between β_1 and β_2;
- we should take the absolute value $|\beta_1 - \beta_2|$, because often β_2 will not be greater than β_1; and
- some notion of scale, perhaps provided by τ, is required.

Camargue measures progress by adapting a metric used by Padberg and Hong [25] to measure the outcome of their TSP computations. Recall from Section 2.3.2 that Padberg and Hong employ

a pure primal cutting plane approach, solving the linear program to optimality if the solution process fails to certify an optimal tour. In view of this, they seek a measure of progress obtained by their TSP solver which takes into account the possible outcomes of an optimal tour or failure to generate cuts. They define VALUE 1 to be the optimal objective value for the degree LP, with VALUE 2 as the optimal objective value obtained at the end of the solution process. With TOUR as the length of the best tour found, they propose to measure success of the program by the quantity

RATIO = (VALUE 2 - VALUE 1) / (TOUR - VALUE 1).

This metric has some nice properties, as elucidated by Padberg and Hong.

> Note that RATIO is zero if no improvement is obtained (e.g. if no constraint was generated), while RATIO is one if the constraint-generation procedure terminates with the optimal tour. RATIO is, of course, always between zero and one and due to taking both differences and a ratio, the measure is invariant under scaling and translating the data. This is of particular importance since a single ratio, e.g. VALUE 2/TOUR, can be made to "look arbitrarily good" by a simple translation of the data (distances) which affects the numerical values of both tour length and VALUE 2, but not the optimal tour nor the linear programming solution vector.

Let us now see to what extent this metric can be adapted to the purpose of measuring primal pivot progress.

At each pivot step, fix again our notation above where β_1 is the previous pivot value, β_2 is the current pivot value, and τ is the tour found. We may then define our own version of RATIO to be

$$\Phi(\tau, \beta_1, \beta_2) = \frac{|\beta_2 - \beta_1|}{\tau - \beta_1}.$$

This preserves some of the desirable properties of Padberg and Hong's measure. For example, if a pivot is an improving tour, then $\beta_2 = \tau$, and $\Phi(\tau, \beta_1, \tau) = 1$. Similarly, two pivots with identical objective value make Φ equal to zero. On the other hand, Φ values greater than one are possible, although slightly rarer. Rearranging the defining equation for Φ, we see that $\Phi(\tau, \beta_1, \beta_2) > 1$ if and only if $\tau - \beta_1 < |\beta_2 - \beta_1|$. In other words, this happens when x^* and x^{**} are farther apart from each other than $\bar{x}$ is from x^*; and certainly this indicates that a good amount of progress is taking place.

Following the discussion above, Camargue uses two tolerance values, Φ_{next} and Φ_{round} to control the loop of separation routines. A round of cutting and pivoting begins with a pivot whose value is recorded as β_0, and also as the initial value for β_1. At a general step of the loop,

if a separation routine returns cuts, we pivot back to $\bar{x}$, add them to the LP, and record the value of the next pivot as β_2. The efficacy of the cuts added is then measured by the function Φ. If $\Phi(\tau, \beta_1, \beta_2) < \Phi_{\text{next}}$, we move on to the next separation routine. Otherwise, we start again with SEC and cut pool separation. A reasonable choice of Φ_{next} is 0.1.

The other issue mentioned at the beginning of this section was a complete round of separation routines in which progress may seem inadequate, even with cuts returned. To get to the end of the loop means that each routine produced a Φ value less than Φ_{next}, as did the final separation routine called. At this point, we measure progress using the value β_0 stored at the beginning of the cutting loop. If β_F is the pivot objective value after adding cuts from the last separation routine, we compute the value $\Phi_F = \Phi(\tau, \beta_0, \beta_F)$. In their words defining RATIO, Padberg and Hong referred to the degree LP objective value VALUE1 as "the objective function value without cuts", calling VALUE2 "the objective function value with cuts". So Φ_F measures a similar quantity, where "with cuts" and "without cuts" are judged relative to the current loop. If $\Phi_F < \Phi_{\text{round}}$, the entire round of cuts is deemed ineffectual and the loop is terminated, indicating that branching may be needed to advance the solution process. Camargue uses 0.01 for Φ_{round}.

Chapter 7

Augment-and-Branch-and-Cut Searches

An Augment-and-Branch-and-Cut search is the means by which the primal cutting plane solution process is embedded in a branch-and-bound search. The term Augment-and-Branch-and-Cut was coined by Letchford and Lodi [19], who also suggest ABC as a pleasant abbreviation. This chapter explores the ABC implementation adopted in Camargue, tailored specifically to primal cutting plane solution of the TSP. We will begin with a review of the work of Letchford and Lodi [19] on ABC methods, as well as the Behle, Jünger, and Liers [3] study on ABC solution of the degree-constrained minimum spanning tree problem (DCMST). After that, this chapter will outline the implementation adopted in Camargue, which departs quite considerably from the two aforementioned studies.

There are three main topics at hand: *variable selection* asks which edge to branch on, *node selection* asks which branching subproblem (i.e., which node of the search tree) to examine, and *enforcement* is the task of enforcing the disjunction required by a particular node. In this chapter we consider a variable selection rule which attempts to adapt strong branching to the primal case, and a node selection rule which tries to prioritize regions of the search tree that contain good tours. Perhaps the trickiest issue in the primal case is the question of enforcement: how should we enforce branches that contradict the resident best tour? In Camargue, the variable and node selection protocols are chosen so as to lay groundwork for the task of enforcing subproblems. Thus, variable selection, node selection, and branch enforcement are intimately related. Before treating any of these in detail, Section 7.2 lays some preliminaries, defining the notion of a *branch tour* for a branching subproblem. Branch tours are essential to the implementation of branch selection, node selection, and enforcement in Camargue.

7.1 Literature Review

Letchford and Lodi's ABC paper [19] is a successor to their work [18] on primal cutting plane methods, essentially seeking to embed the algorithmic framework of [18] in a branch-and-cut search. All their work is based on the following simple observation, concerning difficulties that may arise when adapting branching to the primal case. Here, we consider a fractional pivot vertex x^* adjacent to some best-known solution $\bar{x}$, quoting directly from [18].

> The point is that one wishes to cut off x^* but does not want to cut off the current best known feasible solution $\bar{x}$ (it may be optimal). If one tries to branch in the standard way, by choosing a fractional variable x_i^* and creating two subproblems (one with the constraint $x_i^* = 0$ added and the other with the constraint $x_i^* = 1$ added), $\bar{x}$ will be excluded from one of the two subproblems. This would mean that we had no starting basis on one of the branches.

Their conclusion is simple: "What is really needed is a non-standard branching rule which removes x^* but leaves $\bar{x}$ feasible on both sides."

If, for example, the variable x_i^* should be fixed to one, Letchford and Lodi's idea is to engage in a branch-and-cut search by searching the polytope obtained by taking the convex hull of the current relaxation polytope and the polytope obtained by setting $x_i^* = 1$. The resulting branching rule assumes a fractional solution x^*, and a feasible solution $\bar{x}$ with variables complemented so that $\bar{x}$ is the zero vector:

> Suppose that $0 < x_i^* < 1$. Create two branches, one with the constraint $x_i = 0$ added; the other with the constraints $x_j \leqslant x_i \forall j \neq i$ added.

So explicitly, in the highly likely event that $\bar{x}$ is *not* the zero vector, the same formula above is computed replacing nonzero entries $\bar{x}_i$ with $1 - \bar{x}_i$.

In their computational experiments, Letchford and Lodi suggest traversing the ABC tree in a depth first fashion, always first examining the node which affirms the current solution (i.e., which simply adds the constraint $x_i^* = \bar{x}_i$). This has a helpful consequence for the implementation of their branching rule. Let m be the number of variables in the LP, and let C be the number of affirming branch nodes examined thus far. When considering the first branch that contradicts the current solution, Letchford and Lodi show that the number of branching constraints that need to be added is then $m - C - 1$. Moreover, further contradicting branches can be enforced by changing the sense of an existing branch constraint from inequality to equality. It is also fortunate that each branching constraint contains precisely two nonzero entries, but Letchford and Lodi emphasize that this is only possible under the assumption that all constraints in the LP are tight at the current solution.

Behle et al. [3] adopt the Letchford-Lodi framework for a study on primal cutting plane solution of the DCMST. Their work has been briefly reviewed a couple times in this thesis, most relevantly in Section 4.1.2 where we described their approach to primal inseparable connectivity constraints. In this situation and in general, Behle et al. allow non-tight constraints to be added to their LP relaxations. Letchford and Lodi write that

> [if] it is desired to include non-tight constraints in the LP, then it is still possible to perform the above branching but the non-tight inequalities require some additional 'work'.

Behle et al. carry out this 'work', outlining a formula for branching constraints that requires some more case checking, and a more complicated structure for the branching constraints, with more nonzero entries.

Now is a good time to discuss some reasons why the Camargue approach departs from that proposed by Behle et al. and Letchford and Lodi. The main concern is the number of nonzero entries that need to be added to the LP relaxation to effect a branch which contradicts the current solution. Letchford and Lodi's computational studies [19] are carried out on multi-dimensional 0-1 knapsack problems, using starting constraint matrices with five or ten rows and 5–25 columns. And in the Behle et al. study the test bed consists of DCMST instances with 50 or 100 nodes. This is to say that both authors use a highly nontrivial bed of test cases, but that the number of columns in the models they consider is small compared to even medium-sized TSP instances. In a TSP instance on n nodes, we mentioned in the previous chapter that the size of the core edge set is usually anywhere from $1.5n$ to $5n$ columns in size. Thus, branching in opposition to the current tour means adding on the order of $3n$ to $10n$ nonzero entries to the LP relaxation.

The other, closely-related issue concerns variable pricing. Although Letchford and Lodi's studies [18] and [19] mention the possibility of variable pricing, this topic is not treated in the test cases they consider; and likewise with the experiments of Behle et al. The addition of branching rows, one for each column, seems to become a more complex matter when considering an edge set of frequently-changing size. And as regards edge pricing itself, it is not immediately clear how these branching constraints should be treated when computing reduced costs for edges that are not in the core LP.

I suspect that the adaptation of such branching rules to edge pricing schemes would be a rich and interesting direction of research; this is only to say they fall outside the scope of investigations conducted in this thesis. Moreover, it would be interesting to adopt the branching scheme outlined by Behle et al. and Letchford and Lodi for solutions of TSP instances on sparse edge sets. Such sparse instances could be obtained either by construction, or by using (most likely dual fractional) methods to permanently eliminate variables from consideration. Indeed, we saw in Section 4.3 that this approach was put to good effect by Cook et al. in a study of safe Gomory

cut computation for LP relaxations of the TSP. At this time, however, such branching rules are wholly unimplemented in Camargue.

7.2 Branch Tours and Problem Enforcement

In this section we define the notion of a *branch tour* for a given branch node. Informally, a branch tour is a tour that satisfies all branching constraints at a given branching node. We shall give a more precise definition presently, before discussing strategies for computing such tours. These branch tours are a vital to the enforcement of branching subproblems in Camargue, and indeed to all other aspects of ABC searches.

7.2.1 Definitions and First Properties

Let $G = (V, E)$ be a graph describing a TSP instance, and let b be either zero or one. Given e_i in E, let $e_i(b)$ denote the branching constraint for the edge e_i in direction b. That is, the symbol $e_i(b)$ denotes adding the constraint $x_i = b$ to the current LP relaxation. A branch node N for an ABC search on G is thus defined by a set $F = \{e_{i_1}(b_{i_1}), \ldots, e_{i_k}(b_{i_k})\}$ of edges and branching constraints.

Given a search tree node N with constraint set F, we define a *branch tour* for N, denoted $T(N)$, to be tour which satisfies all branching constraints in the set F. Explicitly, suppose we represent $T(N)$ as a cyclic permutation of vertices $(v_1, v_2, \ldots, v_n)$. Then

- if $e_i = \{s, t\}$, and $e_i(1)$ is in the constraint set for N, then s occurs directly before or after t in $T(N)$; and
- if $e_j = \{u, v\}$, and $e_j(0)$ is in the constraint set for N, then u never appears directly before or after v in $T(N)$.

We will try to develop intuition for branch tours by describing some simple properties and examples.

Let T be the best known tour at the root of an ABC tree, i.e., the best known tour at the beginning of an ABC search. Suppose we branch on the variable x_i, creating two search tree nodes N_0 with constraint set $F_0 = \{e_i(0)\}$ and N_1 with constraint set $F_1 = \{e_i(1)\}$. Then T is also a branch tour for precisely one of N_0, N_1: if $\bar{x}$ is the incidence vector of T, and $\bar{x}_i = b$, then T is a feasible branch tour for the node N_b. More generally, a branch tour $T(N)$ for a search tree node N is always a feasible branch tour for precisely one child node of N.

Another interesting property of branch tours is their relation to the feasibility of branching nodes. In the general exploration of a branch-and-cut search tree, a key task is the identification of *infeasible* nodes: as we add constraints fixing an edge equal to zero or one, it may be the case that such fixings render the LP relaxation infeasible. It is perhaps vacuous to observe, therefore, that a branch node is feasible if and only if it admits a branch tour.

The preceding observation is a bit more useful than it may seem at first glance, due to a slight subtlety in the way we have defined branch nodes and branch tours. Indeed, given a node N with constraint set F, we have spoken of modifying a given TSP relaxation by adding all the constraints in F. The LP relaxation itself has no direct bearing on the branch tour $T(N)$, however, as $T(N)$ was defined as a permutation of nodes.

Note for example that if the edges $e_1, e_2, \ldots, e_k$ all share a common endpoint u, then the constraint set $e_1(1), e_2(1), \ldots, e_k(1)$ is infeasible by *over-fixing* if $k > 2$ because this contradicts the degree constraint $x(\delta(u)) = 2$. For complementary reasons, if G is the complete graph on n vertices, a node which fixes $n - 1$ neighbours of a vertex u to zero is infeasible by *under-fixing*. Of course, in the previous chapter we mentioned that the complete graph edge set is rarely (if ever) present when considering an LP relaxation for a TSP instance. We instead operate with a graph $G_C = (V, E_C)$, where E_C is the core set of edges currently in the LP relaxation. More often than not, a vertex u will have degree much lower than n in G_C, so that a node N may appear infeasible by under-fixing on u despite admitting a branch tour $T(N)$. The search for branch tours may therefore provide a means of recovering the feasibility of a branch node LP, as we describe presently.

7.2.2 Finding Branch Tours

Camargue tries to compute branch tours heuristically using the chained Lin-Kernighan algorithm, as implemented by Applegate et al. in Chapter 15 of [2]. The approach is equal parts simple and imperfect; we describe it now before addressing some of its obvious limitations.

Consider a TSP instance $G = (V, E)$ and edge costs $(c_e : e \in E)$. Suppose we are trying to compute a branch tour $T(N)$ for a node N with constraint set F. Define the graph $G_N = (V, E_N)$ to be obtained from G by deleting all edges from E that are fixed to zero in F. Then, define the edge costs $(c_e^N : e \in E_N)$ such that c_e^N is set to a large negative value for each edge e fixed to one in F. For all other edges f in $E \cap E_N$, the cost c_f^N is just c_f.

Given the graph G_N, Camargue first searches for obvious over-fixing or under-fixing infeasibilities as described in the previous section; this can be done easily by storing for each node v in V a count of the respective number of fixed and restricted edges incident with v. Absent any such infeasibilities, chained Lin-Kernighan is invoked on G_N with $|V|$ kicking steps (see Section 15.3 of [2] for a discussion), with a limit of at least 200 and at most 1000. And from the point of

view of the chained Lin-Kernighan implementation, deleted edges in $E \setminus E_N$ are simply assigned extremely large travel cost, so the branching tour from the parent node of N may be provided as a starting cycle, even though it is infeasible half the time. The tour returned from the procedure just described is then double-checked for compliance with the constraint set F.

Having shown that the process is simple, let us discuss why it is also imperfect. For starters, over- and under-fixing infeasibilities are sufficient but *not* necessary for G_N to admit no branching tour: G_N may not admit branching tours for more complex, subtle reasons. Stronger sufficiency conditions than the one described are certainly possible, but no necessary and sufficient condition is known. And indeed, certifying the existence or non-existence of such cycles is an $\mathcal{NP}$-complete problem which may be formulated as a special case of the TSP. In view of this, it is a conscious decision to favour a relatively fast and efficient heuristic approach in Camargue.

7.2.3 Node Enforcement

We have now laid enough groundwork to describe how branch nodes are enforced in Camargue; and indeed the answer has probably been foreshadowed by the discussion of branch tours, and the discussion of limitations of existing ABC studies. Namely, branching subproblems in Camargue are enforced just as in the dual fractional case, by clamping the bounds of a variable being branched on.

Given a branch node N, Camargue will try to compute a branch tour $T(N)$, which is then set as the starting basic feasible solution from which non-degenerate pivots are computed. At all times we store only the best tour T currently known for the TSP instance, and the tour $T(N)$ for the current node. When we first examine N, all cuts which are slack at $T(N)$ are moved to the cut pool. Conversely $T(N)$ is now the vector used to drive the primal separation algorithms of Chapters 4 and 5, and cuts in the pool which have become tight at $T(N)$ are eligible for re-inclusion in the core LP.

Of course we have only treated the favourable situation in which a branching tour is successfully computed. In the preceding section we point out that this may not always be possible, so a contingency plan is required. Failing the computation of a branching tour, the core LP machinery of Camargue enters a "tourless" mode of operation. In this mode, the resident best tour T is again used as a starting basis for computing pivots, which of course implies the added overhead of pivoting from T to a primal feasible basis. While in tourless mode, we allow the addition of non-tight cuts to the core LP, whether from standard separation heuristics or from the cut pool. Tourless operation persists for as long as the node N is under consideration, or until cutting and pivoting on N identifies a pivot which is a branching tour for N.

This is a dire state of affairs, to be sure. Arguments about cutting off line segments between vertices of a TSP relaxation polytope are certainly much less convincing when one of the polytope

vertices is cut off by a branching constraint. The situation seems to arise relatively rarely in practice, though.

7.3 Variable Selection

The goal of this section is to adapt the technique of *strong branching* to the primal case. Strong branching is a rule which can be used to select general branching structures, such as subtour constraints, in addition to branching edges, although we are concerned presently just with edges. The technique was pioneered by Applegate et al. (see Section 14.3 of [2]) in the context of the TSP, and it has since found fruitful application as a branch selection rule in a wide class of optimization problems.

Strong branching evaluates a variable x_i by temporarily setting x_i to zero or one, performing a limited number of dual simplex pivots, and recording the objective value obtained for both branching directions. The estimation generally occurs in multiple phases: a large number of variables are examined with a relatively small number of pivots, and then a smaller number of variables from this batch are re-evaluated with a greater iteration limit. Note that the initial set of candidates may be computed by evaluating the *Driebeek penalties* on variables, a measure of objective value change that implicitly computes a dual simplex pivot that would take place after fixing a variable.

Indeed, this is the main hurdle that must be overcome when attempting to adapt strong branching to the primal case: by and large, strong branching is implemented as a measure based on the *dual* simplex method, and LP solvers which provide strong branching functionality often do so with the expectation of a resident basis obtained by dual simplex optimization. Of course the primal case does not produce dual solutions and lower bounds, it produces tours and non-degenerate pivots. Tours and pivots, therefore, will be the starting point for our implementation.

Some aspects of the strong branching search will be adapted with little adjustment from the dual fractional TSP case, or from general MIP solution. The first is the notion of an evaluation function for ranking down- and up-estimates, and multiple phases of evaluation. Explicitly, for a variable x_i, a down-estimate z_0 for x_i is obtained by fixing $x_i = 0$, and an up-estimate z_1 is obtained by fixing $x_i = 1$. An estimate pair (z_0, z_1) is then evaluated using a function

$$p(z_0, z_1, \beta) = \frac{\beta \cdot \min(z_0, z_1) + \max(z_0, z_1)}{\beta + 1},$$

with values of β varying based on the phases of the selection procedure. In the Concorde TSP solver [2], Applegate et al. perform strong branching first by computing Driebeek penalties for all fractional basic variables, ranking them with $p(z_0, z_1, 10)$. Strong branching then proceeds in two

phases. From the initial estimates, the K_1 best are evaluated with γ_1 dual steepest-edge simplex pivots, setting $\beta = 100$ in the function above. Then, the best K_2 of these are evaluated with γ_2 dual steepest-edge simplex pivots, also with $\beta = 100$. Typical values are $K_1 = 5$ first-round candidates with $\gamma_1 = 100$ pivots, followed by $K_2 = 2$ second-round candidates with $\gamma_2 = 500$ pivots.

In Section 5.1 of [1], Tobias Achterberg investigates strong branching as a variable selection rule for solving general MIP and constraint integer programming problems in his SCIP software. The SCIP approach differs from that used in Concorde in several ways, and one point of departure we emphasize is that of the iteration limit. SCIP records the average number $\bar{\gamma}$ of simplex iterations used per LP solver thus far, performing a strong branch probe with an iteration limit of $2\bar{\gamma}$. A lower limit of 10 iterations is imposed, with an upper limit of 500.

Camargue adopts the same evaluation function p, with an initial collection of candidates selected by a longest fractional edge criterion (see e.g., Section 14.1 of [2]). Candidates are then evaluated in batches of $K_1 = 5$ and $K_2 = 2$, just as in Concorde. The choice of iteration limits borrows ideas from both Concorde and SCIP. We propose to store the average number $\bar{\varphi}$ of iterations per *non-degenerate pivot* used in computations thus far. The first round of candidates is examined with at most $\gamma_1 = \bar{\varphi}$ pivots, and the second round is examined with at most $\gamma_2 = 3\bar{\varphi}$ pivots. As in SCIP, we insist that at least ten and no more than 500 iterations are performed. The choice of iteration limits means that in the first round we simply hope to emulate a non-degenerate pivot, and in the second round we aim for a deeper probe which gives an idea of the lower bound for the current edge set.

The motivation for the higher second-round iteration limit is to facilitate the possibility of *pruning* branching subproblems based on lower bound. In the dual fractional case, further exploration of a branching node can be avoided given a lower bound which is within one unit of (or greater than) the current best known tour. Letchford and Lodi [19] suggest that in the primal case, a node no longer needs to be searched precisely once the best known tour is optimal for that node. By searching with more simplex iterations, we allow the possibility of a strong branch search which returns an optimal LP solution with objective value that proves the optimality of the current tour. In this case, we can attempt to verify the bound thus obtained using column generation, thereby possibly pruning the subproblem before generating any cuts. Similarly if the branching probe reports that the LP is infeasible, we can try to recover the feasibility of the LP by adding edges, pruning the problem if this is not possible. This is a common-sense approach that can be adopted with little overhead by storing information that arises naturally during these strong branching probes, as we now explain.

Suppose the edge e is being examined by primal strong branching, so that we are carrying out primal pivots with respect to the nodes N_0, N_1 for branching on e. Let $\bar{x}$ be the resident best tour, with x^* as the solution vector at the end of a strong branch probe for N_i, and let c be the

cost function. If x^* is reported optimal and $c^T x^* > c^T \bar{x} - 1$, then we have a pair of upper and lower bounds which suggest that N_i may be pruned from the search tree, with no need for further branching or cutting. Saving the solution vector x^* and its associated basis, we optimize the LP relaxation at N_i, carrying out an edge pricing scan as described in Section 6.3. The actual Camargue implementation uses a slightly more aggressive approach, optimizing the LP whenever an estimate with $c^T x^*$ bigger than $c^T \bar{x}$ is obtained. If the pricing scan (or manual optimization, in the second case) reveals a reduced lower bound for N_i, pivoting and cutting is carried out in earnest.

The final unaddressed issue is that of starting bases for strong branch searches. We propose to use branch tours as described in the previous section, bearing in mind the caveats about absence of tours, or infeasibility of starting bases. Explicitly if N is the active node and we are considering branching on the variable e, then a branch tour $T(N)$ should be used as the starting basis for the probe, if $T(N)$ is known. Whether $T(N)$ exists or not the problem of feasibility is much the same: $T(N)$ only gives a feasible starting bases for one of the subproblems being examined, and if there is no known $T(N)$ then we have a starting basis for *none* of the subproblems being examined. In either case, the idea is to perform a preliminary number of primal pivots, terminating immediately once a primal feasible basis B has been encountered. If γ_i is the prescribed iteration limit for this search round, we can either proceed with the γ_i pivots from B, or record the objective value at B if the number of pivots required to recover primal feasibility exceeds γ_i.

As compared with the dual fractional case, a higher computational overhead may be incurred by the need to recover primal feasibility from infeasible starting bases. Camargue adopts some simple measures to ease this burden somewhat. Since we only perform strong branch searches in rounds of relatively small batch sizes, it is feasible at each round to save the bases obtained when recovering from an infeasible starting point. These can be saved and reused in the next round, avoiding excess repeated computation.

7.4 Node Selection

The final component of the search process to be discussed is node selection: having chosen edges to branch on, and having established a protocol for enforcing branch nodes, which unvisited nodes of the tree should be examined first?

A simple option, mentioned in Section 7.1, is a *depth first search*. In a depth first search we always process a child node of the current node. This rule is easy to implement and it has the advantage that subsequent LP relaxations tend to share structure, so there is less overhead required in moving from node to node. Following Letchford and Lodi [19], we can also choose to proceed from the root node always examining the node whose branching constraints agree with the current

best tour, until the tour is augmented or proved optimal for a node. Thus, along with the guarantee of closely related LP relaxations, this traversal guarantees the existence of a branching tour well into the search process, since the resident best tour remains feasible. Of course, this rule is highly dependent on the best known tour available when the search commences, and its performance can be quite poor unless the tour is optimal or very nearly optimal. Trading simplicity for a bit more power and flexibility, our aim is to retain some of the desirable properties of depth first branching while developing more sophisticated rules in the following section.

7.4.1 Branching by Bounds and Tours

We now consider node selection rules which build on some of the machinery developed in this chapter, using primal strong branch estimates and branch tours to guide node selection.

The first node selection rule, of a *best bound search*, is fairly straightforward. A best bound search prioritizes regions of the search tree estimated to have weak dual lower bound, thereby effecting the greatest increase in the global dual lower bound. This rule (and a more advanced machination) is the node selection rule of choice in the Concorde TSP solver of Applegate et al. [2]. As described in Section 14.3 of [2], the dual strong branch estimates used to rank edges also provide a good estimate of the dual lower bound for subproblems being examined. Having developed our own notion of primal strong branch estimates in the previous section, we are equipped to apply this node selection rule in the primal case as well.

Applegate et al. provide some useful insight on the motivation behind their use of strong branching and best bound searches. They write that

> in a best-bound search, the dominant criterion for preferring one structure over another is the increase obtained in the lower bounds of the child subproblems. Other criteria, such as quickly locating a portion of the search tree that contains a high-quality tour, are of much less importance, given the goal of exactly solving the TSP.

This makes sense because Concorde is, at its heart, a highly effective dual fractional TSP solver which works by rapidly driving up the dual lower of an LP relaxation until reaching an optimal tour. The principles espoused by Applegate et al. ring true in the primal case as well. On the one hand, the "A" in ABC stands for "augment". But while we may be more interested in "locating a portion of the search tree that contains a high-quality tour," we cannot ignore the fact that dual feasibility is what proves the optimality of a primal feasible solution. We keep this tension in mind, using it to motivate the next two node selection ideas.

A *best estimate search* is essentially the primal counterpart to a best bound search. Where best bound searches prioritize good dual solutions, best estimate searches prioritize good primal

feasible solutions. In a best bound search it seems clear that strong branch probes provide a good proxy for dual bounds, but it may be less clear how to predict the quality of primal feasible solutions that can lie in a region of the search tree.

Common approaches (see e.g., Section 6.4 of [1]) involve using a linear estimate which takes into account the best known primal feasible solution, together with fractionality measures for dual solutions at certain nodes. In Camargue we propose to direct best estimate searches in a much more complex way, eschewing linear measures in favour of estimates arising from branch tours. In Chapter 14 of [2], Applegate et al. describe their variable and node selection procedures by saying that they are

> willing to spend a significant amount of computation time in the evaluation procedure, going well beyond the fast selection rules that have been used in previous studies.

It is in this exact spirit that we guide primal best estimate searches.

The result is a procedure we will call a *best tour search*. Recall that we use sparse, chained Lin-Kernighan tours to find a good tour as a basic feasible solution to pivot from when examining a node. Using shorter runs of chained Lin-Kernighan (with half as many kicking steps), we can compute decent-quality branch tours for nodes *before* they are examined, storing their length as a proxy for the tour quality that can be expected at a given node. Like best estimate search, best tour search can be implemented by storing unvisited nodes in a priority queue, ranked by the length of branch tour computed. Although the chained Lin-Kernighan algorithm is immeasurably more complex than computing a linear estimate, it is not a bad bottleneck in practice.

The final approach we consider attempts to strike a compromise between best tour branching and best bound branching, alternating between the two of them. An *interleaved best tour/best bound search* concedes the argument of Applegate et al. about the importance of dual lower bounds to exact TSP solution, while also recognizing that good tours are vital to generating cuts and pivots in a primal cutting plane TSP solver. The design (and name) of this rule is based entirely on the work of Tobias Achterberg, who motivates it in Section 6.6 of [1]. Achterberg refers to this rule as interleaved best estimate/best first search, since his SCIP software solves MIP and constraint integer programs rather than traveling salesman problems. In such an interleaved search, best estimate is used as the primary node selector, with a best bound selection occurring at a fixed frequency. Camargue uses the same parameter as SCIP, making a best bound selection every ten nodes. Achterberg demonstrates the computational efficacy of interleaved best estimate/best bound search in Section 6.8 of [1], and it is used as the default node selection rule for SCIP.

7.5 Computation

We now consider a limited computational study of some of the node selection rules just described, comparing an interleaved best tour/best bound search with a best bound search. Since the next chapter is devoted in earnest to wider computational experiments with Camargue, the goal of these experiments for the most part is to qualitatively profile the impact of different node selection strategies. We consider some studies based around some smaller TSPLIB [28] instances, as well as randomly generated Euclidean instances.

7.5.1 Node Selection and the TSPLIB

The first test bed was pared down from the collection of all TSPLIB instances having less than 500 nodes. For each instance, Camargue was ran once using interleaved best tour/best bound search, using the current time as the random seed. Then, a successive trial was performed with best bound search, using the same random seed from the first trial. An artificial wall-clock time limit of one hour was imposed on the trials.

Both solution methods failed to solve the instances gr431 and pr439 within the one hour time limit; these instances are omitted. Additionally, best bound branching failed to solve d493, terminating after an hour with four unvisited nodes. Interleaved branching solved it in approximately 61 minutes of CPU time, which suggests only a marginal victory.

The remaining instances included are all those where at least one solution protocol required a search depth greater than one. Thus, we omit instances where both selection rules either solved the problem at the root node, or solved it after branching on precisely one edge. These tests are summarized in Figures 7.1 and 7.2.

Figure 7.1 shows for each instance the number of branching nodes that were visited before solving the instance to optimality, and Figure 7.2 shows the CPU time required for each instance. Note that although random seeds are fixed, the parallel implementation of exact blossom and simple DP separation introduces an element of non-determinism to the results. Moreover, no attempt was made to average the results of Figure 7.2 with multiple trials. With all this in mind, we emphasize that CPU time results should be interpreted more qualitatively.

Unfortunately, it would appear that the most decisive conclusion to be drawn from these results is that both node selection strategies are viable. Figure 7.1 shows that, for the most part, best bound search is better at producing smaller search trees, consistent with the results reported by Applegate et al. in Chapter 14 of [2], and by Tobias Achterberg in Chapter 6 of [1]. However this reduction in search tree size does *not* always imply faster solution times, as shown in Figure 7.2. It is especially interesting to note that search trees of identical size were produced for pcb442, but the interleaved search completed much faster.

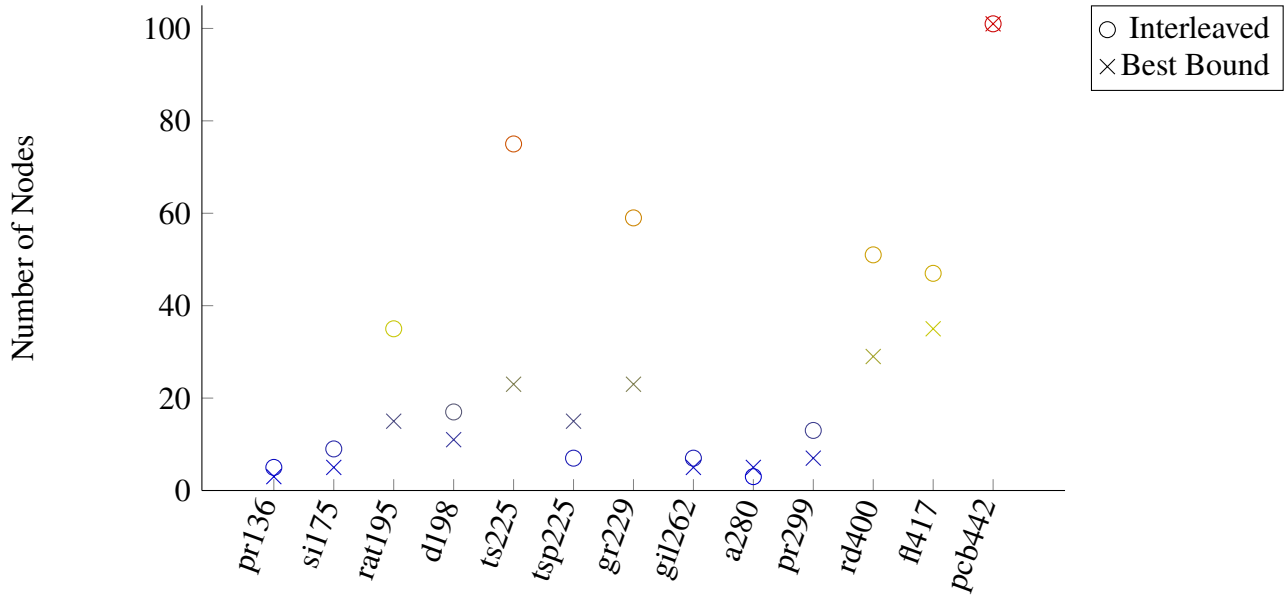


Figure 7.1: The impact of node selection strategies on number of branch nodes.

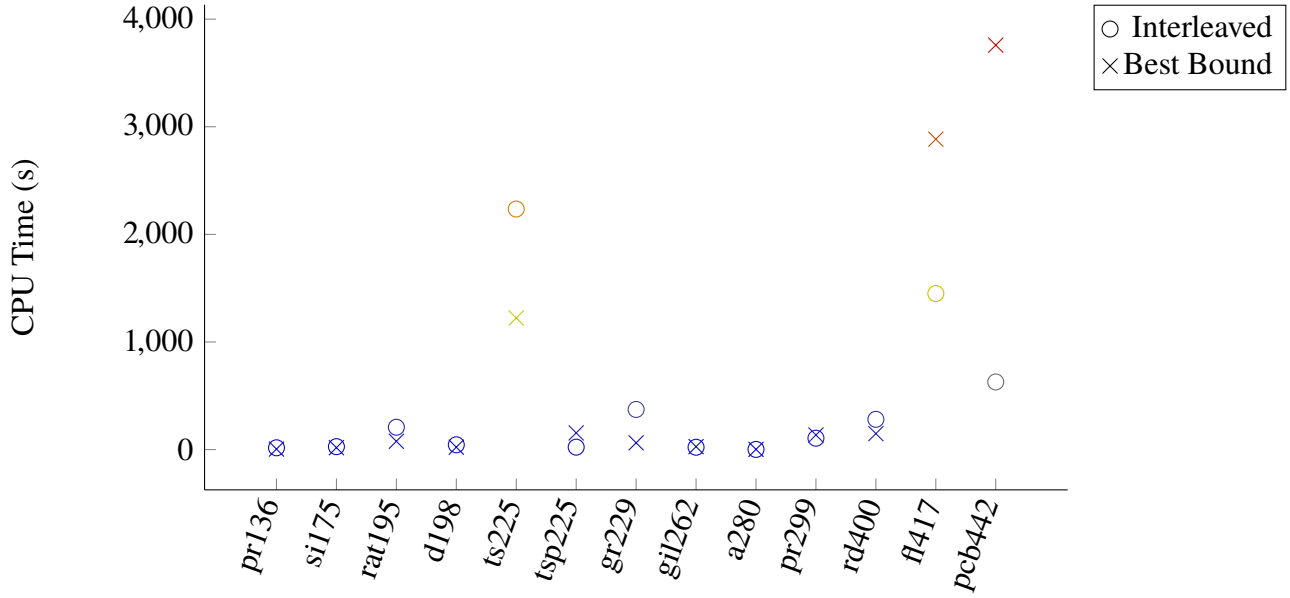


Figure 7.2: The impact of node selection strategies on CPU time.

Again, a full battery of computational tests is postponed until the next chapter, but these results seem to show that Camargue is capable of solving these smaller-sized TSPLIB instances in a reasonable amount of time. For most instances of size 400 and below, optimal solutions are returned in under 5–7 minutes of CPU time, with best bound searches performing slightly better in some cases. The exception is ts225, a challenge problem contrived to be difficult to solve by cutting plane methods. Here, the best bound search performs much better both in terms of solution time and search tree size. On the other hand, the largest two problems suggest an advantage for interleaved best tour/best bound branching.

Case Study: Branch Tours for pcb442

Of all the results reported in Figures 7.1 and 7.2, perhaps the most interesting feature is the discrepancy in solution times for pcb442, despite the identical number of search nodes. This behaviour deserves further examination, and it provides a good opportunity to gauge the impact of branch tour construction on the solution process. Figure 7.3 attempts to explain this discrepancy by plotting branch tour length as a function of depth in the search tree. (Note that the interleaved search attained a greater depth than the best bound search, so there is no data on the best bound search for depth levels 11 or 12.)

The data in Figure 7.3 was computed as follows. As the ABC search progresses, we may examine several different nodes $N_1, \ldots, N_k$ of identical depth in the search tree. For each of these nodes, we compute a branch tour $T(N_k)$ to be used as a starting basis and source for cutting planes. Suppose that, at the time of visiting $T(N_k)$, the global best known tour has length $\tau^*_{N_k}$. If τ_{N_k} is the length of the tour $T(N_k)$, then we can record the ratio

$$\rho(N_k) = \frac{\tau_{N_k}}{\tau^*_{N_k}}.$$

Thus, $\rho(N_k)$ is the ratio of branch tour length to the best tour length at that point in the search tree. For example, a value of $\rho(N_k)$ equal to 1.001 indicates that cuts and pivots were computed at node N_k with a tour whose length was 0.1% longer than the best available tour. Thus, lower values of ρ are better. In particular, values less than one indicate a branch tour which improves on the current best known tour (a phenomenon observed in other test cases, but not this one).

For each depth, and for both node selection protocols, Figure 7.3 records the arithmetic mean of the tour length ratios $\rho(N_1), \ldots, \rho(N_k)$. In words, Figure 7.3 reports the average quality of tours that were available for each depth of the search tree.

Regardless of node selection protocol, Figure 7.3 shows favourable results on the viability of computing branch tours, even while adopting a relatively simple, low-effort strategy. Tours within 99.9% of optimal are common. I will remark that, due to the non-determinism discussed

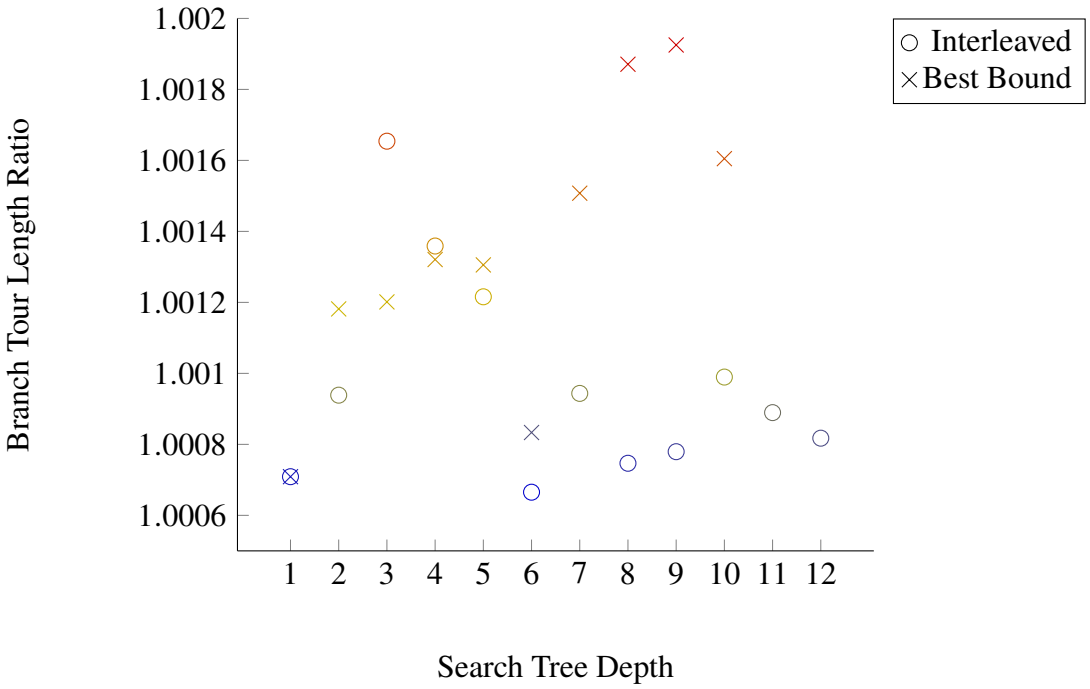


Figure 7.3: Branch tour length ratios for pcb442.

earlier, it was actually the *best bound search* which found an optimal tour sooner, with several augmenting pivots occurring at the root LP, i.e., before any divergence of solution approaches. A tour of the same length was not discovered until depth 3 of the interleaved search tree.

Up to depth 5 or so, results are mixed, without a clear advantage for either protocol. But deeper into the search tree, a greater discrepancy in branch tour lengths emerges; and with it, a possible explanation for the performance discrepancy of Figure 7.2 emerges as well. Indeed, having identified an optimal tour, all that remains is the "BC" in ABC, i.e., the task of certifying the tour's optimality by a dual lower bound. Although this is the very purpose of a best bound search, it seems that the nature of primal cutting plane computation makes this a more difficult prospect when pivoting and cutting with a tour of worse quality. While this result seems to support interleaved best tour/best bound branching, it may also just suggest the need for more robust methods of constructing branch tours, regardless of node selection strategy.

7.5.2 Node Selection and Random Problems

The experiments conducted thus far consider TSP instances of fairly small size. On many of these instances, the initial chained Lin-Kernighan tour is optimal or extremely close to optimal, so we do not get a sense of the impact of node selection rules on tour augmentations. We now consider an additional set of experiments, conducted on random Euclidean instances having 1,000 nodes.

For each of the node selection rules, we consider a batch of ten randomly generated Euclidean instances, created one after the other using the current time as the random seed. Thus, unlike the previous section, node selection rules were *not* used on identical instances, or with identical starting edge sets.

Table 7.1 reports for each node selection protocol the performance of Camargue on the bed of test cases. The column "Trial" indicates which trial from 1 to 10 was being performed. The "Augs" column counts the number of augmenting pivots that took place, and the "Nodes (CMR)" category records the number of nodes in the search tree after Camargue was terminated, either by optimality or by time limit. For the "Nodes (CC)" column, each instance was then passed to Concorde, and we record the number of search tree nodes required for Concorde to solve the problem to optimality. This column therefore is meant to give a sense of how difficult each instance is. Finally, for instances that were not terminated after the 2-hour time limit, the column "CPU Time (s)" reports the number of seconds of CPU time for the solution, run on a server with two Intel Xeon 4-core 3.5 GHz CPUs.

The most obvious conclusion to be drawn from Table 7.1 is that 1,000 node random Euclidean problems pose a significant challenge to the Camargue solver, regardless of node selection strategy. The interleaved search solved only two instances to optimality, just one more success than obtained with the best bound search. At face value, an 80% failure rate isn't much more or less exciting than a 90% failure rate, so we will try to analyze performance based on CPU times and Concorde search tree size.

For both protocols, the only problems that were solved to optimality appear to be ones that were easy for Concorde to solve, given the relatively small number of search nodes. Of course this is not a perfect metric of problem difficulty, but the easiest problem appears to have been Trial 2 solved by the best bound search, where Concorde returned an optimal solution at the root node. The best bound search failed to solve Trial 1, which required five nodes for Concorde. Conversely, five Concorde nodes matches the dimension of the interleaved search for Trials 3 and 10, where an optimal solution was certified. Moreover, the best bound search required a relatively large CPU time and search tree size to solve Trial 2 to optimality.

Trial 5 for the interleaved search appears to be the hardest problem of any in the test set, requiring nearly 1,000 branch nodes for optimal solution by Concorde; in all likelihood Camargue never stood a change at solving this problem. On the other hand, Trial 3 gives the most exciting

Trial	Augs	Nodes (CMR)	Nodes (CC)	CPU Time (s)
Interleaved Best Tour/Best Bound				
1	7	105	53	–
2	9	81	13	–
3	3	7	5	336.31
4	3	49	29	–
5	3	79	971	–
6	9	57	35	–
7	4	43	75	–
8	5	73	15	–
9	2	81	9	–
10	0	9	5	1910.61
Best Bound				
1	8	77	5	–
2	2	19	1	2057.91
3	3	47	33	–
4	2	39	61	–
5	4	19	3	–
6	11	27	31	–
7	8	19	141	–
8	1	31	25	–
9	2	55	13	–
10	6	61	59	–

Table 7.1: Performance of node selection protocols on 1,000 node random instances.

results out of any from this experiment, with an optimal solution reported in about six minutes of CPU time with seven search nodes. The case with Trial 10 is quite different, however. Note the entry of zero in the "Augs" column, indicating that the starting tour was optimal. As compared with Trial 3, Trial 10 required two more search nodes and about six times as much CPU time to certify the optimality of the initial tour.

In view of the infrequency with which optimal solutions are certified, and the somewhat harsh 2-hour time limit, we consider a final analysis of the performance of Camargue in terms of the quality of tours returned at termination. Indeed, the defining characteristic of primal solution methods is that they yield a sequence of improving tours, whereas dual fractional methods yield a sequence of improving lower bounds. Figures 7.4 and 7.5 attempt to grade the node selection criteria from this point of view. When passing each instance to Concorde for generating the

data in Table 7.1, the optimal tour lengths were recorded for instances that were not solved to optimality by Camargue. Moreover, in all the trials of Camargue we keep a record of every augmenting pivot obtained, along with its objective value. Thus, Figures 7.4 and 7.5 show for each node selection criterion the progression of tour quality through the solution process. In each figure, we superimpose the tour trajectories for all trials with each separate node selection rule. The horizontal axis indicates the number of augmenting tour vectors found: the 0^{th} tour is the starting chained Lin-Kernighan tour, and positive values delineate a sequence of augmenting tour pivots. For each such tour vector, the vertical axis shows the ratio of tour length to the optimal tour length found by Concorde (or Camargue in the successful trials). Finally, the results for the figures are superimposed in Figure 7.6, although it may be harder to parse this figure without Figures 7.4 and 7.5 as reference.

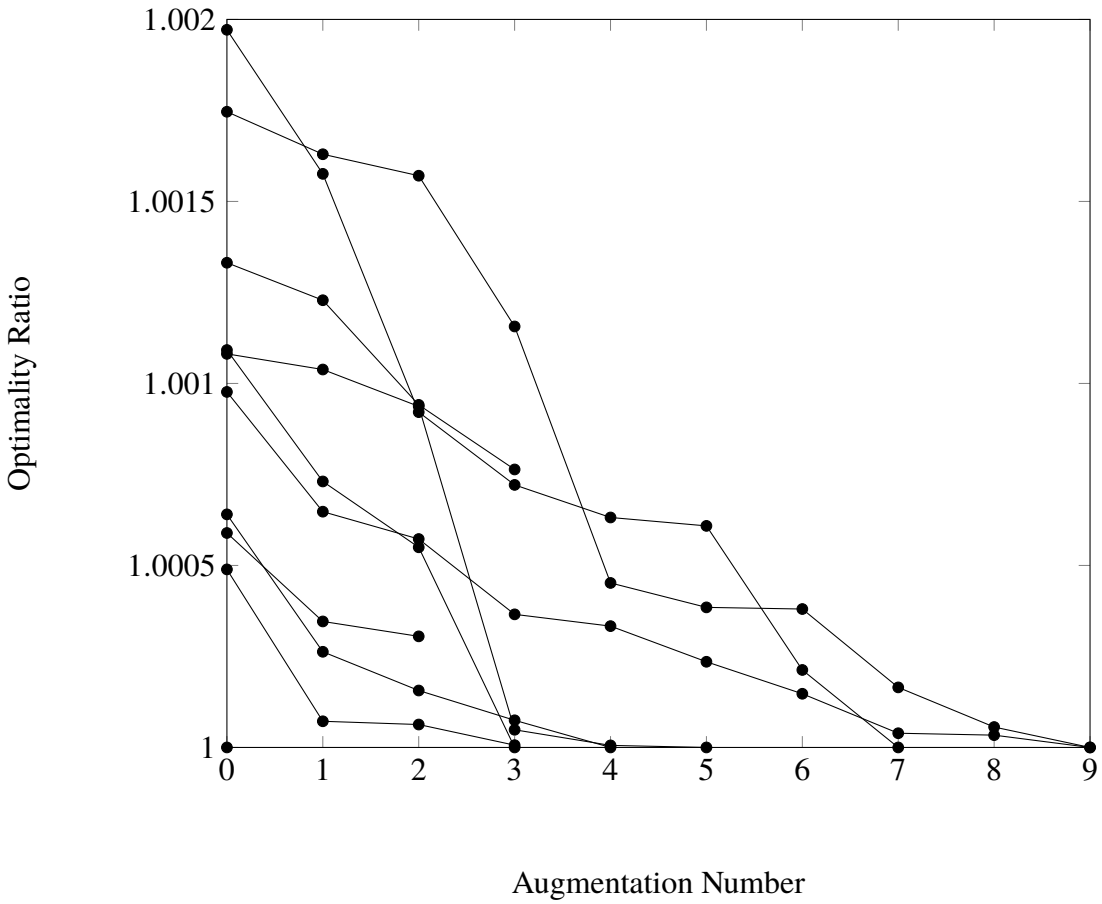


Figure 7.4: Tour length ratios for interleaved search.

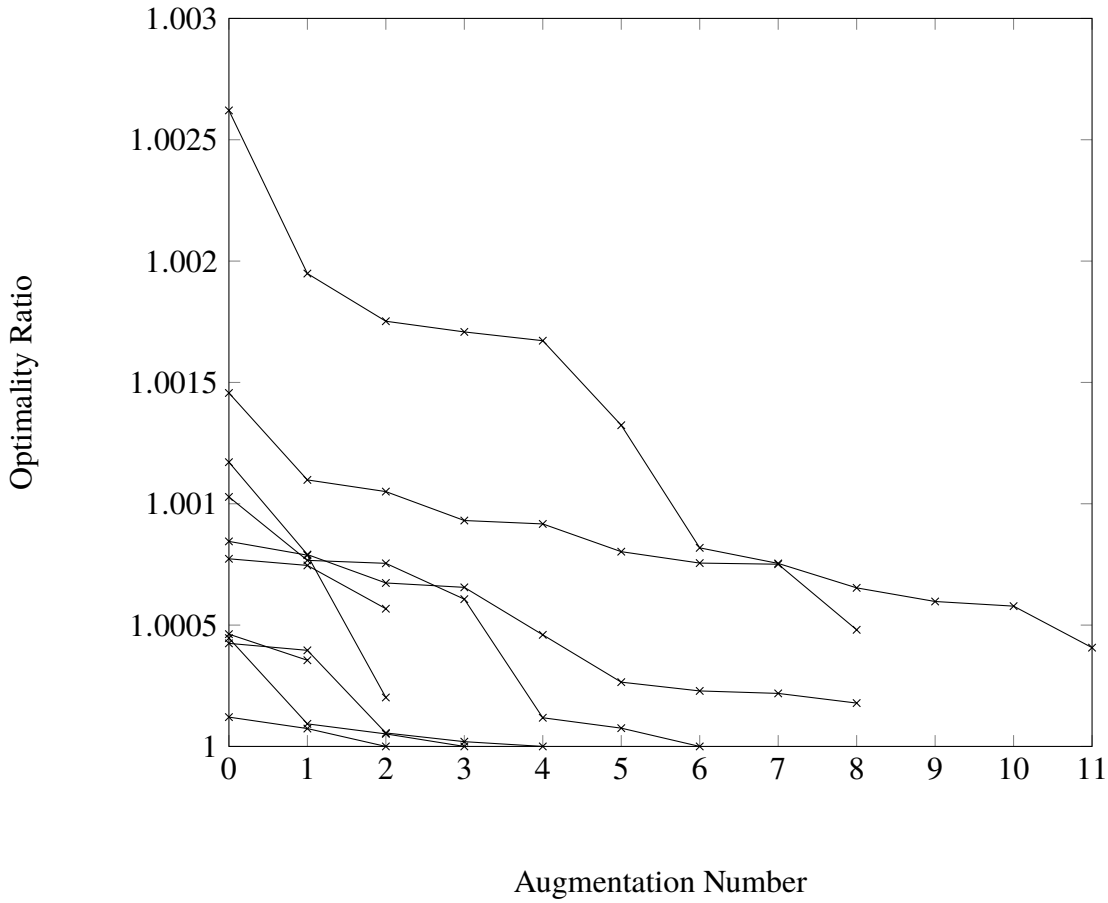


Figure 7.5: Tour length ratios for best bound search.

These figures perhaps finally illustrate a more compelling reason for preferring the interleaved best tour/best bound search to a pure best bound search, at least in the primal case. The interleaved search finds better tours faster, attaining optimality more often. In particular, a close look at Figure 7.4 shows that an optimal tour was attained in six out of ten trials, although one of them is Trial 10 from Table 7.1, in which the starting tour was optimal. By comparison, Figure 7.5 shows only four cases where a tour of optimal length was found. And we can see from the superimposed Figure 7.6 that, although the best bound search sometimes found a greater number of augmenting pivots, their quality was relatively poor.

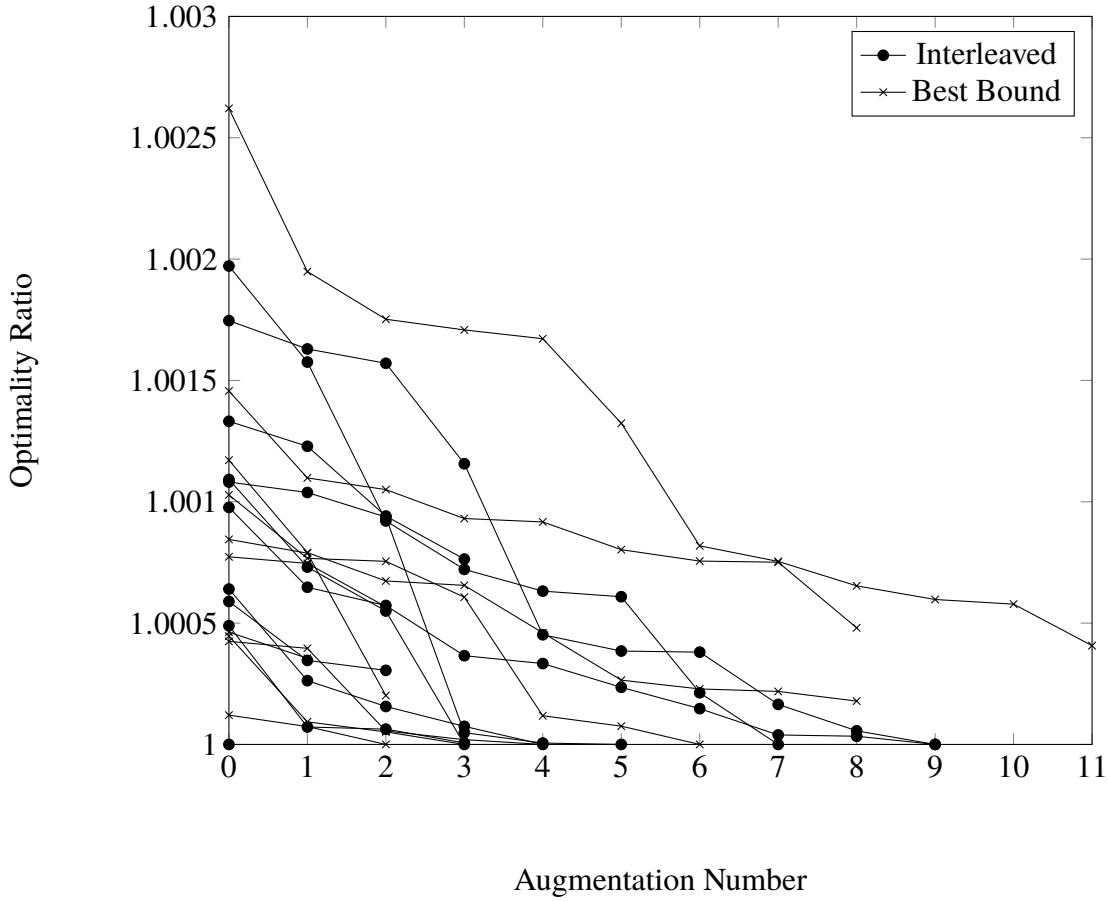


Figure 7.6: Tour length ratios for both search rules.

7.6 ABC Control Flow and Data Structures

At this point we have described all the core components for guiding an ABC search in Camargue. We now give a big picture point of view on the way that these components are used in collaboration, and some of the data structures used to store information about the search.

An ABC tree in Camargue is stored in a simple recursive representation, with each node containing a pointer to its parent, and the root node as the unique node with no parent. Nodes can be classified coarsely as visited or unvisited. At a finer level, a visited node is one that has been pruned by optimality, infeasibility, or one which was split into further subproblems. Unvisited nodes are indicated as needing either pricing, cutting, or branching. We also store the depth of each node, along with objective value estimates from strong branch probes, and the length of a

tour found in computing branch tour estimates. Additionally, a node may optionally hold a basis for a strong branching estimate that appears optimal or infeasible, to facilitate edge pricing.

Camargue implements all of the node selection rules described in the previous section. The interleaved best tour/best bound rule is the default, but others may be selected at execution. A depth first search is managed in an extremely lightweight fashion: no additional data structures are maintained other than the tree representation of the ABC search. For the priority-based rules, such as best tour and best bound, we also keep a priority queue of pointers to unvisited nodes, pushing and popping from the queue as the search progresses. An interleaved best bound/best tour search is implemented much the same way, keeping track of the number of nodes that have been examined so as to invoke the occasional best bound selection.

An ABC search is initiated after a run of primal cutting and pivoting which fails to return cuts, or in which the progress from a round of cuts is too small. As described in Section 6.4, we primal optimize the root LP, using the last non-degenerate pivot as a starting basis, and then try to eliminate variables from the root LP by reduced cost. If some variables can be eliminated, we cut and pivot at the root LP some more before branching begins.

Edge selection is done with primal strong branch probes, proceeding in rounds with fewer candidates and more iterations. We branch on the "winning" edge, adding its unvisited child nodes to the search tree, and computing a quick branch tour for each node, recording only the tour length. If no tour can be computed, a large positive value is recorded. In this way, regardless of node selection rule being employed, we keep the information used for all of the rules described in the previous section. The idea, also suggested by Tobias Achterberg in Chapter 6 of [1], is to use complementary estimates as tie-breakers within a node selection rule. For example if a node is being chosen by best bound, then

- among nodes with the same bound, we prefer the one of greater depth in the search tree; and
- among nodes with the same bound and search tree depth, we prefer ones with better tours.

Thus, even when not performing interleaved searches, we implicitly blend different node selection rules. The idea of breaking ties by depth is to add something resembling a *plunging* strategy to the node selection rules, in which we prefer to examine unprocessed children of a current subproblem. This idea is not implemented in a more rigorous way, but as mentioned at the beginning of the last section it can help generate sequences of closely-related subproblems, making it easy to modify the LP relaxation and compute branch tours.

Cutting and pivoting on nodes themselves proceeds much as described throughout this thesis, and in particular the ideas of the previous chapter are used to measure progress of pivoting, and to manage cuts and variables. And before the cutting and pivoting process begins, we compute a

branch tour to be instated at the active node. As discussed, longer runs of chained Lin-Kernighan are used here, as compared to the estimating procedure. In some admittedly rare cases, a tour may be found where one was not found during the estimate. It may also quite rarely happen that a branch tour of length *less* than the global best tour will be computed, in which case this tour is used to update global data that is used to manage the search.

Chapter 8

Computation

All of the algorithms and ideas in this thesis are implemented and tested in the Camargue computer code, a TSP solver. As a whole Camargue is not capable of reliably solving large-scale TSP instances, but all of its parts have been designed with such instances in mind. The guiding philosophy for the Camargue code, and of the experiments in this thesis, has been to approach the TSP both as a rich problem in its own right, and as a vehicle for designing and testing techniques of combinatorial optimization.

In this chapter we will evaluate the performance of Camargue on a variety of test cases, considering TSPLIB instances and random Euclidean problems. While the main object of concern is exact solution of the TSP, an attractive aspect of primal cutting plane algorithms is the prospect of obtaining good-quality solutions even if optimality is not attained or certified. Our computational results will be evaluated with respect to both these criteria.

8.1 The Camargue Code

The development of Camargue was closely tied to the writing of this thesis, guiding the experiments, implementation choices, and computational results reported in previous chapters. It is freely available online at

https://github.com/cstratopoulos/camargue.

The code itself is about 15,000 lines written in the C++ programming language under the ISO C++11 standard. Camargue has been developed more or less in tandem with this thesis, and its development is still an ongoing process, As such, some of the more limited test results from earlier chapters have been conducted with earlier versions of the code. However, all results in

this chapter, and the branching tests from the previous chapter, have been conducted with the Camargue Release v0.3 code accessible at the link above, where it is also possible to browse changes from earlier versions.

Camargue was implemented by opting at all times to make use of existing codebases and libraries to the greatest degree possible: it uses the CPLEX 12 callable library for all linear programming computation, and the Concorde callable library is used extensively as well.

Concorde is the golden standard of TSP computation, and comparisons between Camargue and Concorde are only natural. It is for this reason I emphasize that Camargue makes *extremely* widespread use of a large range of facilities implemented by Concorde. Just to give a sense, Camargue uses the Concorde callable library to compute

- chained Lin-Kernighan tours;
- initial edge sets;
- edge pricing scans;
- primal separation of segment cuts;
- standard separation of connected component SECs, fast blossoms, block combs, cut metamorphoses, and local cuts;
- minimum cuts in separating primal exact blossoms;
- segment cuts for finding candidate teeth;
- odd cuts in separating simple DP inequalities;

and so on. And in areas where Concorde code is not used directly, it is often the prototype for implementations adopted in Camargue. Thus, Camargue should be regarded as something of an overgrown offshoot of Concorde, complementing a proper subset of Concorde's features with various methods and implementations that are particular to the primal case.

8.1.1 Initial Tours and Heuristics

At various points in this thesis we have considered computational experiments giving a narrow view of the Camargue solution process and control flow. In particular Chapter 6 gave an overview of the construction of starting edge sets, and the control of cutting and pricing. Chapters 4 and 5 gave an account of the separation routines themselves, and by default Camargue employs all

separation routines described therein, including the more involved standard heuristics. We now name explicitly some of the settings and heuristics employed for constructing and finding tours in Camargue.

Camargue opts for a more aggressive approach to starting tour computation, as compared with the Concorde solver for example. Camargue first computes ten independent chained Lin-Kernighan tours with a random starting cycle, and the best of these is then used as the starting cycle for an additional chained Lin-Kernighan tour. The best tour thusly found is then set as the starting basis for cutting and pivoting, and it is kept permanently as the reference tour for storing compressed hypergraph vertex sets.

The identification of augmenting pivots is also aided by a simple *rounding heuristic*, employed whenever the solution process fails to return primal cutting planes. If x^* is the last non-tour pivot encountered, we try to construct a tour from x^* by adding its edges to a tour vector in a greedy fashion, proceeding from edges e with x^*_e closest to one. The resulting tour is then used as a starting cycle for a quick chained Lin-Kernighan tour. Primal rounding heuristics are a well-studied tactic in general integer programming, but the idea and implementation of this particular heuristic is found in the Concorde TSP solver source code.

This rounding heuristic sometimes gives a way to jump-start a stagnant cutting and pivoting loop, adding another means by which augmenting tour pivots can be identified. From instance to instance the source of augmentations can vary considerably. In some cases almost all augmentations are due to this rounding heuristic, but in other cases it does not appear at all. Either way it is an efficient heuristic that is called fairly infrequently, so it is a natural choice for inclusion in the Camargue code. It is disabled only when an ABC search begins, to avoid the extra effort required to ensure that such tours are compliant branch tours.

8.2 Exact Solution

In this section we will consider tests focused on random Euclidean instances as well as TSPLIB instances. Here, the foremost criterion of concern is the ability of Camargue to obtain and certify optimal tours for its inputs. That is, we are testing the capabilities of Camargue as an exact TSP solver. Following the discussion and experiments of the previous chapter, all trials are conducted with the interleaved best tour/best bound search as the branch node selection protocol.

Our tests will follow two protocols: for random instances and for TSPLIB instances below 1,000 nodes, a time limit of four hours is imposed on computations; this is restrictive, but less so than the limit from the branching experiments of Section 7.5.1. This time limit reflects the standard, mostly driven by Concorde, that instances of size at most 1,000 are considered as being of “small” or “modest” size. Following this, we consider some TSPLIB instances of size at least

1,000 where Camargue is given free reign to search for solutions with no explicit time limits.

8.2.1 Random Euclidean Instances

The following set of tests considers random Euclidean instances of fixed dimension: ranging from 100, 200, 300, 400, 500, we perform ten separate trials, using the current time as a random seed to generate a scattering of points placed in a uniform random fashion on a grid with horizontal and vertical coordinates ranging from 0 to 1,000,000 - 1. In the case of smaller instances with very quick solution times, artificial gaps between trials are imposed to ensure that the same instance is not identically generated twice in a row. Distances between cities are defined using the EUC_2D norm of the TSPLIB, which rounds the familiar Euclidean norm to the nearest integer. All tests were performed on a server with two Intel Xeon 4-core 3.2 GHz CPUs.

Randomly generated instances are a rich source of nontrivial test cases in TSP computation, and the use of repeated trials gives a good way to profile the variance in solution speeds that can occur even within problems of a fixed size. Thus, in Table 8.1, we attempt to report a moderate range of summary statistics for all the instances considered. To start, we report in the "Successes" and "Failures" column the number of problems that were solved to optimality in the 4-hour time limit. After that, "CPU seconds" reports the geometric mean of CPU seconds required to solve each instance. Problems that did not solve in the specified time limit are treated as having contributed the full four hours of CPU time to the mean. In a sense this gives an advantage to the failed instances, since four hours is a strict lower bound on the solution time that would be required. On the other hand, it allows useful information to be returned from a bed of test cases that did not all complete successfully, and it seems reasonable given the typical solution times for instances that *did* solve within the time limit.

Moving along, the second half of the table reports a "three-number summary" for the number of branch nodes in the ABC searches conducted. For all ten separate trials, we report the minimum, median, and maximum number of search tree nodes used to solve the instance. Again, as with the case of CPU times, search tree sizes for instances that failed to solve are still recorded, taking the number of nodes in the tree at termination. Unlike the case of CPU times there is little compromise to the integrity of the data, if any: problems in this test bed only ever hit the time limit after branching at the root LP, in which case the "Maximum" column almost always corresponds to the search tree size for an instance that was prematurely terminated.

There is a lot of interesting information to be gleaned from Table 8.1. The expected trend of solution times increasing with problem size is present, although it is interesting to note that there is almost no change between solution times for 200 and 300 nodes, with a sharp jump from 300 to 400. Similarly, the portion of failures grows with problem size as well, although for 500 nodes and under we still have almost all instances solving to optimality within the specified time limit.

Cities	Successes	Failures	CPU seconds	Branch Nodes		
				Minimum	Median	Maximum
100	10	0	1.14629	1	1	3
200	9	1	20.4269	1	1	421
300	10	0	20.6377	1	1	23
400	9	1	212.263	1	9	153
500	7	3	1108.05	1	27	155

Table 8.1: Running times and summary statistics for random Euclidean instances.

Note also that for all instances, a minimum search tree size of 1 is present, indicating at least one problem which was solved at the root node with no branching. This is the median search tree size for instances below 400 nodes as well, indicating that most trials solved without any branching required. A small median is still present with 400 nodes, with a big jump from 400 to 500 nodes. And it is interesting to note that the maxima for the 400 and 500 node problems are fairly close.

An extremely bizarre trial emerged for one of the 200 city random problems, the only instance under 400 nodes to terminate due to the time limit. The search tree grew to an extremely large size, with 27 unvisited problems remaining at termination. To give an idea, the second largest number of nodes in the overall search tree for an instance of size 200 was 29. Examining the log of computation for this failed trial gives a good hint of what went wrong.

After branching on an edge e at the root node, the tour branching estimate for the down branch indicated a significant advantage over the up branch, and this side of the tree was probed to depth five before one of the nodes was pruned by optimality. At this point, the next problem in the queue to be examined was the up branch on e, at which point the additional number of kicks in the branch tour search identified an augmenting branch tour, i.e., one which improved on the best known at that time. When exploring this side of the tree, good quality branch tours were obtained and nodes at depth as low as three were able to be pruned by optimality. At this point however, the search returned to descendants of the down branch at e. The new optimal tour provided a very poor quality starting cycle for branch tours on this side of the tree, and at greater depths of the search tree tours of quality only within 98% of optimal were found, which is quite terrible by the standard we are used to.

In computing branch tour estimates we suggested in the last chapter that half as many kicking steps could be used when compared to the tours actually being constructed for instatement at a given node; but it seems that this conservative approach may cause more problems than it solves. If the optimal tour had been known before exploring the down branch at e, it is likely that the search could have been pruned sooner, before descending into a region of the tree where relatively bad-quality tours were available. This also suggests that it may be useful to compute

branch tours with *denser* edge sets as compared with the active edges of the core LP, such as quad-nearest-neighbour edges, or perhaps the Delaunay triangulation of the point set. The sheer fact of having spent over four hours of CPU time on a computation for a 200 node instance shows that such costly approaches would probably be justified. Another fruitful approach could be to explore data compression methods that allow full or partial edge sets of branch tours to be stored for various regions of the search tree, in this way providing better starting cycles for tour computation.

8.2.2 The TSPLIB

No study on TSP computation is complete without tests on the TSPLIB [28] test set, a collection of problems curated by Gerhard Reinelt. TSPLIB instances have been used for many of the experiments and illustrations presented in this thesis, and we now aim to systematize this study to a greater degree, developing on the branching experiments of Section 7.5.1.

The first set of test cases we consider consists of a single trial with all TSPLIB instances having under 1,000 cities. Of the 76 instances in this set, all but three solved within the four hour time limit. The exceptions were pa561, d657, and u724, all problems on the larger end of the spectrum. All experiments were performed on a server with two Intel Xeon 4-core 3.5 GHz CPUs. In Figure 8.1 we report a log-scale scatter plot of CPU time in seconds, based on problem node count.

Note that the CPU times only include the primal cutting, pivoting and branching process, not the time to construct initial tours or edge sets. Moreover, times are recorded to a relatively low precision, with some smaller instances reporting a zero second solution time; this should be understood to mean that construction of the initial tour and edge set took the longest out of any aspect of the computation, and that the actual solution completed in less than 0.01 CPU seconds.

Figure 8.1 shows the expected trend of CPU time increasing with problem size. But there are some interesting outliers which deserve further comment. For example, all instances under 100 nodes solved in less than three seconds of CPU time, with the exception of pr76, which took around 75 seconds. For comparison, all other instances under 200 nodes are solved in under 65 seconds of CPU time, with times of five to ten seconds being the most common. In my experiments with cut selection routines, it appears that cut metamorphoses are critical to fast solution on this instance. With only SECs, blossoms, block combs, and simple DP cuts, solution times in the range of five to eight minutes were more common.

The other obvious outlier is ts225, an instance to which we devoted special attention in the studies of Section 7.5.1; it requires a number of CPU seconds comparable to instances two to three times as large. In this particular trial, the solution required nearly three hours of CPU time,

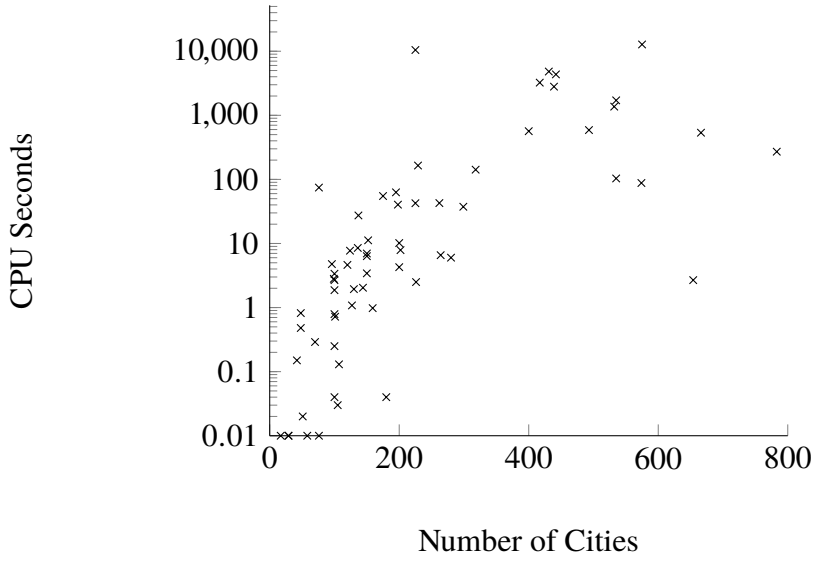


Figure 8.1: TSPLIB instance solution times under 1,000 nodes.

exceeding the solution times for all other instances except rat575. This is also a large degradation over the 37 minute solution time reported in our branching experiments.

On the other hand not all of the outliers are pathological: it is interesting to note that p654 solved in only two seconds of CPU time, a speed completely out of range of the other instances of comparable size. Additionally the largest instance solved, rat783, took less than five minutes of CPU time. On the whole, these time-limited experiments show that Camargue is certainly capable of tackling small TSP instances, although it does not seem to produce results in a speedy or robust fashion.

Finally, we now consider a limited set of TSPLIB examples of size at least 1,000 nodes. Camargue is allowed here to run without an artificial time limit, giving a better sense of its performance on these midsized examples. All tests were performed with a single trial, again on a server with the same specifications as the smaller TSPLIB instances. Table 8.2 reports results for some of the instances considered. The "Branch Nodes" column reports the number of nodes in the search tree, with "CPU Seconds" reporting CPU time for the single trial performed.

The results of Table 8.2 show that, unfettered by artificial time limits, Camargue is indeed capable of solving some medium-sized instances to optimality. It is exciting to see that both pr1002 and si1032 were solved with no need for branching, and with relatively short solution times: si1032 finished in just over a minute, with about eight minutes for pr1002.

With the other instances the solution time is much greater, falling in a range around five to

Problem	Branch Nodes	CPU Seconds
dsj1000	41	28198.76
pcb1173	179	33286.26
pr1002	1	447.93
si1032	1	73.04
u1060	131	20391.53
vm1084	27	30703.75

Table 8.2: Tests on some medium-sized TSPLIB instances.

nine hours of CPU time. But it is interesting to note that sprawling branch trees are not necessarily the culprit. For example the instance u1060 required around five times as many branch nodes as vm1084, while solving 1.5 times faster.

A closer look at computation logs indicates that overgrown edge sets may have had more to do with the long computation times, with some instances having a final edge set of size around 10–18 times the number of nodes. This appears to be due to a subtle issue with edge pricing scans performed during the ABC search, at times when tours appear optimal at a given branch node. Likely more care is required in pruning cuts from the LP relaxation when going from node to node, and with computing lower bounds that account for branched edges.

8.3 Tour Improvement

The results of the preceding sections show that, while problems in a neighbourhood of 1,000 nodes are not completely out of reach for Camargue, they pose a significant challenge for its facilities at present. We now engage in experiments of drastically different character, testing the ability of Camargue to produce good quality tours under constrained time limits. We now describe the motivation and design of these tests.

Letchford and Lodi [18] make an interesting parenthetical remark when motivating and characterizing primal cutting plane algorithms. They write the following.

> The goal of the primal method is to iteratively improve the solution . . . by moving from one vertex . . . to another, until no more improvement is possible and optimality is proven. (It can be viewed as an exact version of the well-known local search heuristic.)

From a practical point of view, this may be an attractive feature of primal cutting plane methods. Namely, a prematurely-terminated course of primal cutting and pivoting may leave a better

quality tour, along with a possibly weak lower bound. By comparison, early termination of a solver like Concorde will yield a drastically improved dual lower bound, but the starting chained Lin-Kernighan tour may have been improved upon little if at all. Thus, we can consider the utility of Camargue from the point of view of a use case which seeks good quality tours in a relatively short period of time, using invocations typical of tour-finding heuristics. This results in something of a "chained Camargue" approach, as we now describe.

We first perform three independent runs of Camargue, setting a time limit of ten minutes. We do these runs with no edge pricing and a pure primal cutting plane approach. This implies in particular that safe Gomory cuts can be used in these trials, which finally gives us a chance to test the machinery developed in Section 4.3. To say that the runs are independent means that each time a separate random seed is used in construction of the initial edge set (consisting of the union of 10 chained Lin-Kernighan tours as described in Chapter 6), and for the chained Lin-Kernighan tour that is used as the actual starting tour.

The best tour found in this way is then used as a starting tour for another invocation of Camargue, setting a time limit of one hour. And this time, the Delaunay triangulation is used as the initial core edge set. The idea here is that this denser edge set provides a better proxy for the complete graph edges, perhaps improving the chance of identifying augmenting tours. Edge pricing is still not performed, but branching is permitted on these trials; we use a pure best tour branching search, with no attempt to interleave the best bound estimates.

In all these trials, moreover, we provide the known optimal tour length as a lower bound which can be used to terminate the search. That is, identification of a tour with objective value matching the optimum will cause early, successful termination of the solution process. The use case that we model again resembles tour-finding heuristics, in which a user may specify a target objective value with the understanding that they would accept tours of length better or equal to this target. Thus, each of the individual trials may be terminated either by objective limit, time limit, failure to find cuts for the pure primal trials, or if the tour is optimal for the chosen sparse edge set.

We will consider tests using a limited sample of TSPLIB instances, of size greater than the ones completed in the previous section. The test cases are chosen with no particular criteria other than personal interest or fascination. For example, we include d2103, an instance reported by Applegate et al. in Section 16.3 of [2] to require an inordinate amount of time for their Concorde solver. In Section 16.1 of [2] the authors also report the interesting case of the instance u2319:

> the challenge in u2319 is to find the correct tour, rather than finding a good set of cutting planes, since its optimal value is equal to the bound given by its subtour relaxation.

We also consider pr2392, pcb3038, fnl4461, and pla7397, and usa13509. These are just some

larger instances in the TSPLIB which have at some time or other represented landmarks for TSP computation.

Tests results are reported in Table 8.3. The table is divided into columns by trial: "Pre 1" through "Pre 3" indicate the short initial trials with no branching, and "Main" reports the longer trial where limited branching was permitted. Within these, we report under "Aug" the number of augmenting pivots that occurred, and the column "Imp%" measures the percent improvement in tour length that was obtained for that run. Explicitly, it is the difference in length between the initial and final tour divided by the length of the optimal tour. Finally, the "Best %" column expresses optimal tour length as a percentage of the best tour that was found by the procedure. Note that columns with "Aug" equal to 0 and "Imp%" equal to 0.00 indicate that no improvement was gained in that trial.

	Pre 1		Pre 2		Pre 3		Main		
Instance	Aug	Imp%	Aug	Imp%	Aug	Imp%	Aug	Imp%	Best%
d2103	3	0.17	2	0.03	7	0.85	3	0.03	99.76
u2319	0	0.00	0	0.00	0	0.00	1	0.04	99.92
pr2392	1	0.01	3	0.20	1	0.07	4	0.09	99.94
pcb3038	0	0.00	1	0.14	2	0.04	1	0.04	99.74
fnl4461	2	0.04	1	0.04	2	0.04	13	0.07	99.87
pla7397	0	0.00	1	0.16	6	0.16	0	0.00	99.71
usa13509	0	0.00	1	0.07	0	0.00	1	0.08	99.54

Table 8.3: Tour finding results on larger TSPLIB instances.

Table 8.3 suggests that Camargue is capable of making good improvements to tours in a relatively efficient matter, but also that its behaviour can vary dramatically between individual trials. In d2103 for example each trial has a number of augmentations take place, but the most dramatic improvement was obtained in "Pre 3". Of course, quality of augmentations is more important than quantity. In fnl4461 for example 13 augmenting tours were found in the main trial, but the percent improvement was only 0.07. Almost all of the final tours are within 0.3% of optimal, with some even within 0.1% of optimal. The exception is usa13509, but it is also the largest instance reported by a wide margin. Still, the performance is not bad considering that we made no effort to scale the time limits based on problem size.

In Chapter 15 of [2], Applegate et al. comment on the average quality of tours that can be expected with chained Lin-Kernighan, writing that it "can easily produce solutions that are within 1% of an optimal tour". Speaking of the more advanced LKH [12] variation of this heuristic, they write that "this can be lowered to 0.1%". In particular, they report tests in which ten separate trials of the LKH heuristic return a tour for usa13509 with objective value within 0.01% of optimal;

although a much greater amount of computation time is required as compared with the trials we describe here. This "chained Camargue" approach could certainly be adapted to include a greater number of longer pre-trials, but if something more precise than a time limit is preferred, it is perhaps less clear how to adapt the progress metrics of Section 6.5 to the task of tour finding.

Of course it is unclear whether Camargue could reliably find tours of comparable quality to heuristics like LKH, or even more advanced variations thereof. But for example, in Section 16.4 of [2] Applegate et al. describe attacking very large TSP instances with the help of an initial starting tour computed using variations of the LKH heuristic with large amounts of CPU time. I suspect it could be possible to put Camargue to similar use as a tool to aid a more robust, powerful dual-fractional based TSP solver like Concorde, with the advantage that such a solution process also naturally produces a collection of cuts which could be saved for further use in a pool, or something to that effect.

8.4 Conclusions

The TSP has been the subject of enduring fascination in literature, and this is due in large part to the rich structure that it offers a problem worth solving in its own right. But from another point of view, it has also been highly successful as an engine of discovery for techniques in combinatorial optimization, techniques which are often fruitfully applied to other diverse classes of optimization problems. Cut separation, column generation, branching, and heuristic methods are important in all discrete optimization problems, but the implementation of a successful TSP solver requires a great level of sophistication in all these areas.

As regards the task of solving the TSP exactly, the results in this thesis are respectable in some areas, while also leaving a great deal of room for improvement. The greater success, in my opinion, has been the opportunity to adapt a wide range of advanced dual-fractional-based cutting plane methods to the primal case. Invaluable groundwork was laid by the work of Dantzig, Fulkerson, and Johnson [8]; Padberg and Hong [25]; and the various findings of Letchford and Lodi [18], [20], [19]. And moreover, the work of Applegate et al. [2] has been a rich guiding source of methods to adapt, while also setting a lofty target in terms of the scope and scale of problems to be attacked.

In the TSP it seems to be easier to find good tours than it is to find good lower bounds, so that ultimately there is genuine reason to prefer a dual fractional solver like Concorde, used in cooperation with highly effective tour-finding (i.e., primal) heuristic algorithms like LKH. For a wide range of other problems, however, the superiority of dual fractional methods is far from certain. In particular, the study of Behle, Jünger, and Liehers [3] on the degree-constrained minimum spanning tree problem shows that primal methods can be competitive with

dual fractional approaches; and certainly there may be other problem classes of this kind. The results of this thesis show that primal methods have potential as a means for exact solution of the TSP, while also revealing new approaches for attacking other classes of hard combinatorial optimization problems.

References

[1] Tobias Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2009.

[2] David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[3] Markus Behle, Michael Jünger, and Frauke Liers. A primal branch-and-cut algorithm for the degree-constrained minimum spanning tree problem. In *International Workshop on Experimental and Efficient Algorithms*, pages 379–392. Springer, 2007.

[4] Robert E. Bixby, Mary Fenelon, Zonghao Gu, Ed Rothberg, and Roland Wunderling. MIP: Theory and practice – closing the gap. In *System Modelling and Optimization*, pages 19–49. Springer, 2000.

[5] Alberto Caprara and Matteo Fischetti. {0, 1/2}-Chvátal-Gomory cuts. *Mathematical Programming*, 74(3):221–235, 1996.

[6] William Cook, Sanjeeb Dash, Ricardo Fukasawa, and Marcos Goycoolea. Numerically safe Gomory mixed-integer cuts. *INFORMS Journal on Computing*, 21(4):641–649, 2009.

[7] William Cook, Daniel G. Espinoza, and Marcos Goycoolea. Computing with domino-parity inequalities for the traveling salesman problem (TSP). *INFORMS Journal on Computing*, 19(3):356–365, 2007.

[8] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 8:393–410, 1954.

[9] D. de Caen. An upper bound on the sum of squares of degrees in a graph. *Discrete Mathematics*, 185(1–3):245–248, 1998.

[10] Lisa K. Fleischer, Adam N. Letchford, and Andrea Lodi. Polynomial-time separation of a superclass of simple comb inequalities. *Mathematics of Operations Research*, 31(4):696–713, 2006.

[11] Ralph Gomory. An algorithm for the mixed integer problem. Technical report, DTIC Document, 1960.

[12] Keld Helsgaun. An effective implementation of the Lin–Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[13] Torsten Inkmann. *Tree-based decompositions of graphs on surfaces and applications to the Traveling Salesman Problem*. PhD thesis, Georgia Institute of Technology, 2007.

[14] Michael Jünger, Stefan Thienel, and Gerhard Reinelt. Provably good solutions for the traveling salesman problem. *Mathematical Methods of Operations Research*, 1994.

[15] Richard M. Karp. Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane. *Mathematics of Operations Research*, 2(3):209–224, 1977.

[16] Adam N. Letchford. Separating a superclass of comb inequalities in planar graphs. *Mathematics of Operations Research*, 25(3):443–454, 2000.

[17] Adam N. Letchford and Andrea Lodi. Polynomial-time separation of simple comb inequalities. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 93–108. Springer, 2002.

[18] Adam N. Letchford and Andrea Lodi. Primal cutting plane algorithms revisited. *Mathematical Methods of Operations Research*, 56(1):67–81, 2002.

[19] Adam N. Letchford and Andrea Lodi. An Augment-and-Branch-and–Cut framework for mixed 0–1 programming. In *Combinatorial Optimization–Eureka, You Shrink!*, pages 119–133. Springer, 2003.

[20] Adam N. Letchford and Andrea Lodi. Primal separation algorithms. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1(3):209–224, 2003.

[21] Shen Lin and Brian W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[22] Denis Naddef and Emmanuel Wild. The domino inequalities: Facets for the symmetric traveling salesman polytope. *Mathematical programming*, 98(1-3):223–251, 2003.

[23] Manfred Padberg and Martin Grötschel. Polyhedral computations. In E. Lawler, Jan Lenstra, A. Kan, and D. Shmoys, editors, *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization*, pages 307 – 360. Wiley InterScience, 1985.

[24] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large scale symmetric traveling salesman problems. *SIAM Review*, 1991.

[25] Manfred W. Padberg and Saman Hong. On the symmetric travelling salesman problem: A computational study. *Mathematical Programming Study*, 1980.

[26] Manfred W. Padberg and M. Ram Rao. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research*, 7(1):67–80, 1982.

[27] Vincent Raymond, François Soumis, Abdelmoutalib Metrane, Mehdi Towhidi, and Jacques Desrosiers. Positive edge: A pricing criterion for the identification of non-degenerate simplex pivots. *Strathprints Institutional Repository*, page 171, 2011.

[28] Gerhard Reinelt. TSPLIB–A traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[29] Mehdi Towhidi, Jacques Desrosiers, and François Soumis. The positive edge criterion within coin-or's clp. *Computers & Operations Research*, 49:41–46, September 2014.